IMPLEMENTATION

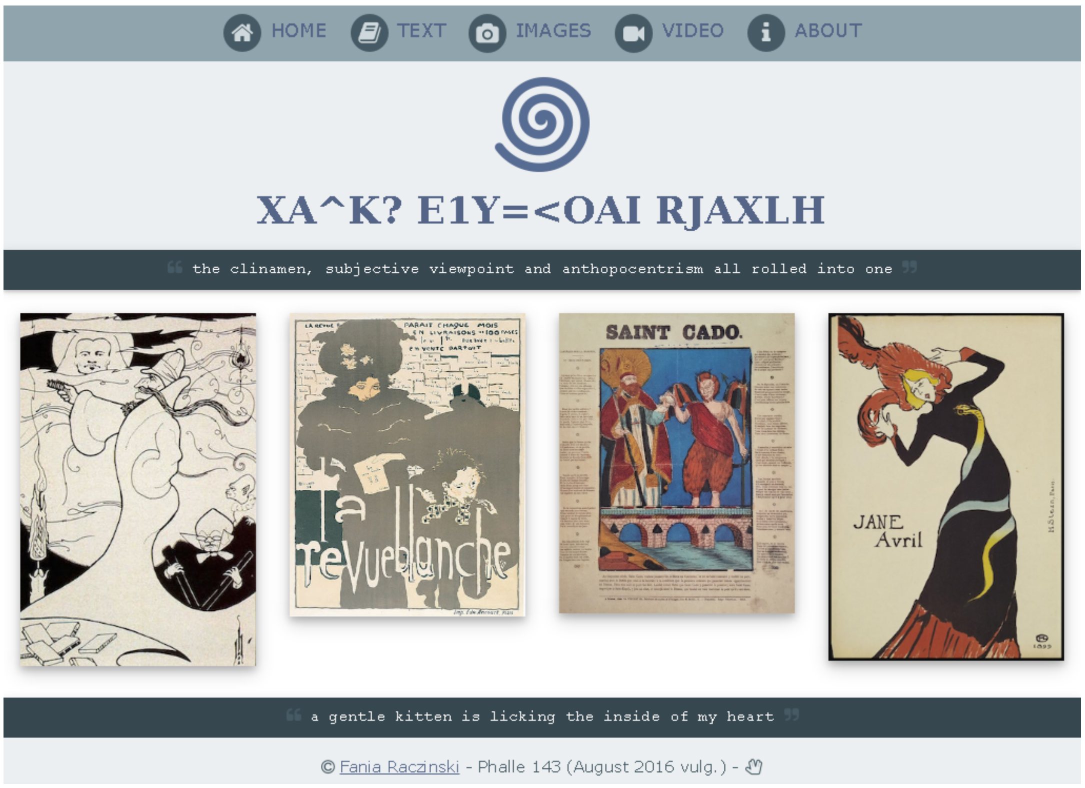
1

Craft against vice I must apply,
you will compel me then to read the will,
this man so complete,
for when thou gav'st them the rod.

A saw a flea stick upon Bardolph's nose,
god may finish it when he will,
deserved thy beauty's use,
you do surely bar the door upon your own liberty.

My heart thy picture's sight would bar,
and finish all foul thoughts,
to dark dishonour's use thou shalt not have.

Their ruth and let me use my sword,
my bare fists I would execute,
is the young Dauphin every way complete.



the clinamen, subjective viewpoint and anthopocentrism all rolled into one

a gentle kitten is licking the inside of my heart

© Fania Raczinski - Phalle 143 (August 2016 vulg.) -

Figure 1.1 – Screenshot of pata.physics.wtf[1]

The website http://pata.physics.wtf (see image **??**) embodies the knowledge

??

```
pata.physics.wtf
├── app
│   ├── static
│   │   ├── corpus
│   │   │   ├── faustroll
│   │   │   ├── shakespeare
│   │   │   └── quotes.txt
│   │   ├── css
│   │   │   ├── dw_glidescroll.js
│   │   │   ├── fania.css
```

[1]The individual letters of the title scramble into place when first loaded. Once this has happend, the title would read: 'PATA.PHYSICS.SEARCH'.

of this doctoral research and showcases **AMC!** (**AMC!**) and patalgorithms. This chapter gives an overview of the structure of the website and the development process.

A high level view of the site would be that it is a pataphysical search engine that subverts conventional expectations by recombining literary texts into emergent user directed and ephemeral poetical structures or unpredictable spirals of pataphysicalised visual media.

It is written in 5 different programming languages[2], making calls to 6 external web services[3], in a total of over 3000 lines of code[4] spread over 30 key files.

Typically, software development is divided into so-called front- and back-ends. The front-end includes web design and web development and is meant to provide an interface for the end-user to communicate with the back-end which involves a server, an application and a database (although this is not fully the case in this project).

??

The front-end design uses the *W3.CSS* stylesheet (**w3css**) as a basis. The website is mostly responsive (see image **??**), meaning it can be viewed well on phones, tablets and desktop screens (the poems and image spirals for example unfortunately have a fixed width which does not scale down well). The site contains various scripts written in **JavaScript** (e.g. scramble letters, randomise poem, send email and tabbed content).

The backend relies heavily on a **Python** (**pythonmain**) framework called **Flask**

[2]Python, **HTML!** (**HTML!**), **CSS!** (**CSS!**), Jinja, JavaScript
[3]Microsoft Translate, WordNet, Bing, Getty, Flickr and YouTube
[4]2864 lines of code, 489 lines of comments - as of 08 Dec 2015

(**Ronacher2016**). Most of the code is written in Python although some parts require a specific templating language called **Jinja** (**Jinja2016**) which renders content into **HTML!**. The application uses several **API!**s (Microsoft Translator, Bing, YouTube, Flickr, Getty and WordNet (**TranslatorAPI**; **BingAPI**; **YouTubeAPI**; **FlickrAPI**; **GettyAPI**; **Princeton2010**; **NLTK2016**)) and is version controlled using **Git** (**Git2016**).

The folder structure is shown in figure **??**. Each spot represents one file. **??**

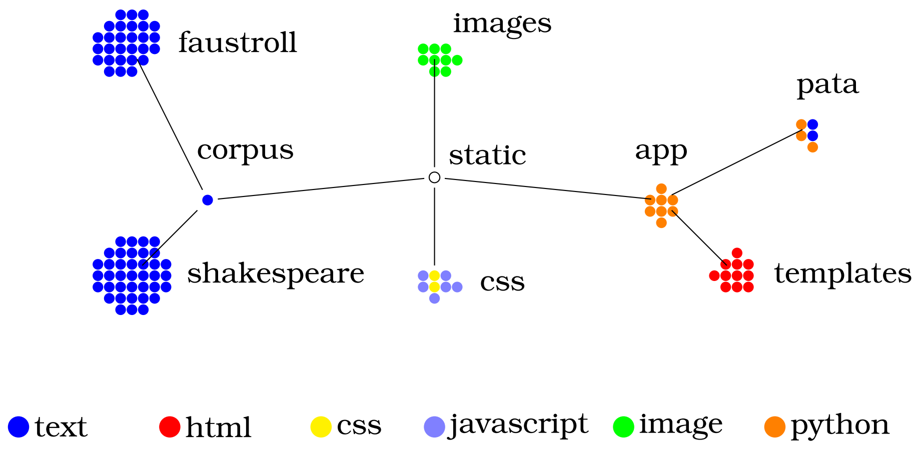


Figure 1.3 – Folder structure and file types

Figures **??** and **??** show the two main workflow scenarios of pata.physics.wtf in the form of sequence diagrams. The columns are labeled with the main agents (this includes the user and the various main files responsible for key actions in the system). Going down vertically represents time.

Figure **??** demonstrates an outline of how the text search process works. A **??** user enters a query into a search box in the `text.html` file which is rendered by the `textviews.py` file. Then it gets forwarded to the `textsurfer.py` file which then handles the pataphysicalisation process and returns patadata back to `textviews.py`. This python file then passes it on to the `textresults.html` file which retrieves and renders the results to the user. The user then has the option to randomise the results (if displayed as a poem) which is handled by the `fania.js` file. A very similar process is in place for image and video search as shown in figure **??**. The main difference is the results are retrieved in the **??** `fania.js` file rather than the `imgresults.html` file.

Putting it another way, (1) the system setup tokenises each of the source texts, removes stopwords and then adds terms and their location to the index (see section **??**), (2) a query then triggers the three pataphysical algorithms, (3) each § **??**
algorithm finds results for the query (see section **??**), (4) some words before/after § **??**

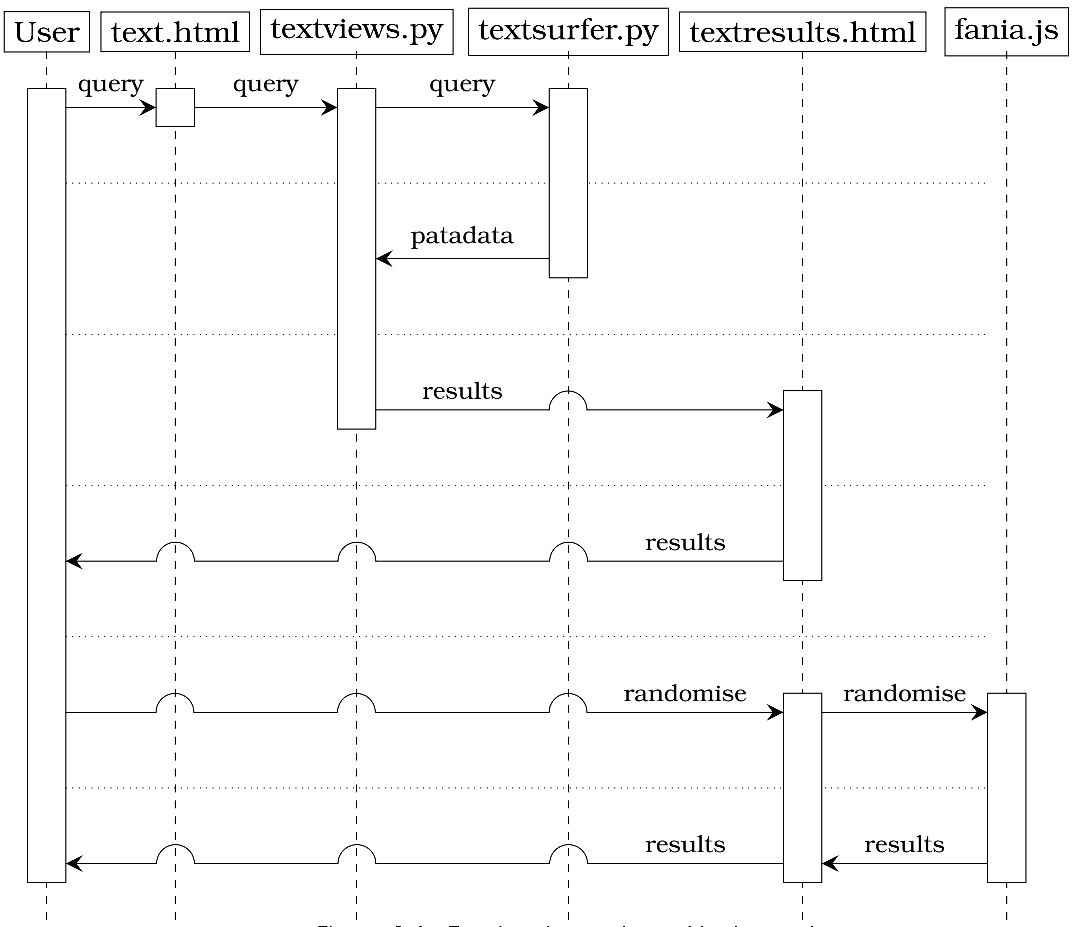


Figure 1.4 – Top-level overview of text search

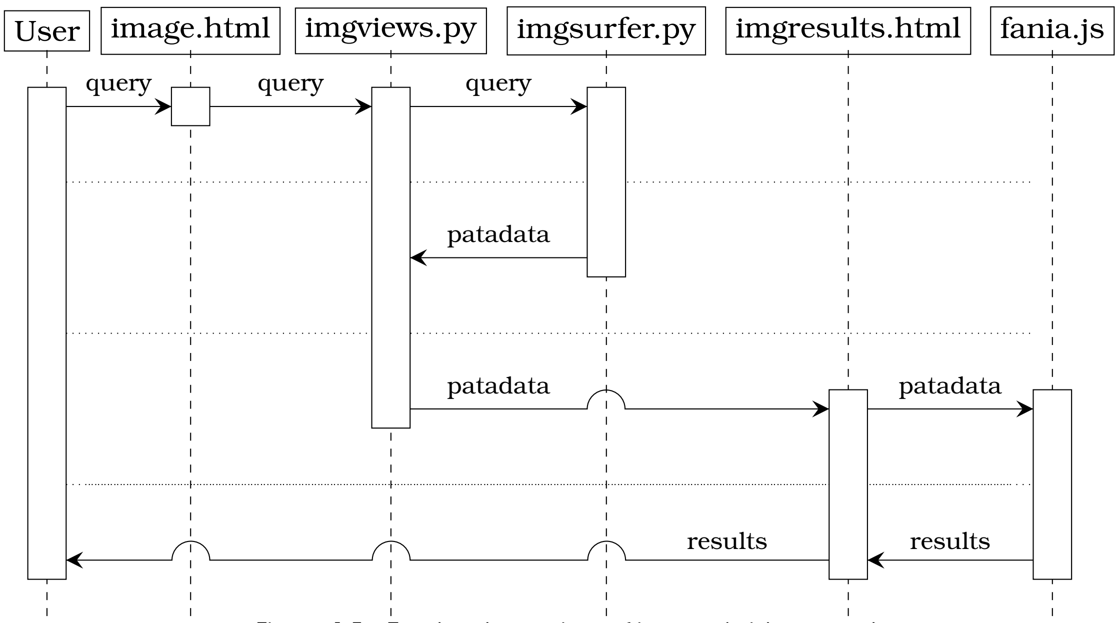


Figure 1.5 – Top-level overview of image / video search

the match are retrieved for context, and (5) the resulting sentences are rendered for the user.

௮ ௮ ௮

The following sections discuss the initial setup of the system when it is first started up, the text search algorithms, the image and video **API!** (**API!**) calls and the main design elements (text poetry and image spirals).

1.1 Setup

The Python web framework Flask (**Ronacher2016**) looks after loading and rendering the various pages for `pata.physics.wtf` (home, text-search, text-results, image-search, image-results, video-search, video-results, about and errors), which means most of the backend related code is written in Python. Although Flask contains a small development server, in a production environment a more capable server is needed. For this reason the Flask site runs on a Gunicorn server (**Gunicorn2016**) and is hosted on a UNIX machine.

1.1.1 Corpora

Instead of crawling the Internet `pata.physics.wtf` uses a local collection of texts for its text search. Setting up a custom web crawler would require a lot more resources (in terms of hardware, time and money) than practical for this project. There are two corpora containing 65 text files together.

The first corpus resembles the fictional library of ‘equivalent books’ from Jarry’s *Exploits and Opinions of Dr. Faustroll, 'Pataphysician* (**Jarry1996**). In principle the corpus is just a folder within the tool’s directory structure containing the following files:

0. Alfred Jarry: *Exploits and Opinions of Dr. Faustroll, 'Pataphysician* (**Jarry1996**)
1. Edgar Allen Poe: *Collected Works* (**Poe2008**)
2. Cyrano de Bergerac: *A Voyage to the Moon* (**Bergerac2014**)
3. Saint Luke: *The Gospel* (**Bergerac2014**)
4. Léon Bloy: *Le Désespéré* (French) (**Bloy2011**)
5. Samuel Taylor Coleridge: *The Rime of the Ancient Mariner* (**Coleridge2013**)
6. Georges Darien: *Le Voleur* (French) (**Darien2005**)
7. Marceline Desbordes-Valmore: *Le Livre des Mères et des Enfants* (French) (**Desbordes2004**)
8. Max Elskamp: *Enluminures* (French) (**Elskamp1898**)

9. Jean-Pierre Claris de Florian: *Les Deux Billets* (French) (**Florian2012**)
10. *One Thousand and One Nights* (**Lang2008**)
11. Christian Grabbe: *Scherz, Satire, Ironie und tiefere Bedeutung* (German) (**Grabbe1995**)
12. Gustave Kahn: *Le Conte de l'Or et Du Silence* (French) (**Kahn2016**)
13. Le Comte de Lautréamont: *Les Chants de Maldoror* (French) (**Lautreamont2011**)
14. Maurice Maeterlinck: *Aglavaine and Sélysette* (**Maeterlinck1918**)
15. Stéphane Mallarmé: *Verse and Prose* (French) (**Mallarme2003**)
16. Catulle Mendès: *The Mirror* and *la Divina Aventure* (English and Spanish) (**Mendes1910**; **Mendes2013**)
17. Homer: *The Odyssey* (**Homer1999**)
18. Joséphin Péladan: *Babylon* (EMPTY FILE)[5]
19. François Rabelais: *Gargantua and Pantagruel* (**Rabelais2004**)
20. Jean de Chilra: *L'Heure Sexuelle* (EMPTY FILE)**??**
21. Henri de Régnier: *La Canne de Jaspe* (EMPTY FILE)**??**
22. Arthur Rimbaud: *Poesies Completes* (French) (**Rimbaud2009**)
23. Marcel Schwob: *Der Kinderkreuzzug* (German) (**Schwob2012**)
24. Alfred Jarry: *Ubu Roi* (French) (**Jarry2005**)
25. Paul Verlaine: *Poems* (**Verlaine2009**)
26. Emile Verhaeren: *Poems* (**Verhaeren2010**)
27. Jules Verne: *A Journey to the Centre of the Earth* (**Verne2010**)

§ **??**

The original list as it appears in 'Faustroll' is shown in chapter **??**. Three of the items have not been found as a resource. Some others have been approximated by using another text by the same author for example. Most of these were sourced from ***Project Gutenberg*** (**Gutenberg2016**) in their original languages. The decision to get foreign language texts was partially due to the lack of out-of-copyright translated versions and partially because the original library in 'Faustroll' was also multi-lingual.

A note on copyright: UK copyright law states in section 5 that the duration of copyright for "literary, dramatic, musical or artistic works" is "70 years from the end of the calendar year in which the last remaining author of the work dies" (**Copyright2015**). Maurice Maeterlinck and Marguerite Vallette-Eymery (a.k.a. Rachilde or Jean de Chilra) died less than 70 years ago and their work should still be under copyright. Alfred Jarry in the Simon Watson Taylor translation is a derivative work and is probably also still protected. However, copyright does not apply when used for "private and research study purposes" as stated in section 7 on *Fair dealing* of (**Copyright2012**).

[5] I have not been able to find any source texts online.

The second corpus is a collection of 38 texts by William Shakespeare (**Shakespeare2011**).

1. *The Sonnets*
2. *Alls Well That Ends Well*
3. *The Tragedy of Antony and Cleopatra*
4. *As You Like It*
5. *The Comedy of Errors*
6. *The Tragedy of Coriolanus*
7. *Cymbeline*
8. *The Tragedy of Hamlet, Prince of Denmark*
9. *The First Part of King Henry the Fourth*
10. *The Second Part of King Henry the Fourth*
11. *The Life of Kind Henry the Fifth*
12. *The First Part of Henry the Sixth*
13. *The Second Part of Henry the Sixth*
14. *The Third Part of Henry the Sixth*
15. *King Henry the Eigth*
16. *King John*
17. *The Tragedy of Julius Caesar*
18. *The Tragedy of King Lear*
19. *Love's Labour's Lost*
20. *The Tragedy of Macbeth*
21. *Measure for Measure*
22. *The Merchant of Venice*
23. *The Merry Wives of Windsor*
24. *A Midsummer Night's Dream*
25. *Much Ado About Nothing*
26. *The Tragedy of Othello, Moor of Venice*
27. *King Richard the Second*
28. *Kind Richard III*
29. *The Tragedy of Romeo and Juliet*
30. *The Taming of the Shrew*
31. *The Tempest*
32. *The Life of Timon of Athens*
33. *The Tragedy of Titus Andronicus*
34. *The History of Troilus and Cressida*
35. *Twelfth Night or What You Will*
36. *The Two Gentlemen of Verona*
37. *The Winter's Tale*
38. *A Lover's Complaint*

1.1.2 Index

When the server is first started various setup functions (such as the creation of the index) are executed before any **HTML!** is rendered. The search algorithms are triggered once a user enters a search term into the query field on any of the text, image or video pages.

</> **??** Each plain text file in the corpus is added to the internal library one by one. Source **??** shows how this is done. The `PlaintextCorpusReader` is a feature of the **NLTK!** (**NLTK!**) Python library (**NLTK2016**) for **NLP!** (**NLP!**). The `words` function tokenises the text, i.e. it splits it into individual words and stores them as an ordered list.

```
library = PlaintextCorpusReader(corpus_root, '.*\.txt')
l_00 = library.words('00.faustroll.txt')
l_01 = library.words('01.poe1.txt')
...
l_27 = library.words('27.verne.txt')
```

Code 1.1 – Adding text files to the corpus library

</> **??** The `setupcorpus` function (see source **??**) is called for each of the text files in the two corpora to populate the index data structures `l_dict` (for the Faustroll vocabulary) and `s_dict` (for the Shakespeare vocabulary).

```
dict = dictionary { dictionary { list [ ] } }
```

A dictionary in Python is what is known as an ‘associative array’ in other languages. Essentially they are unordered sets of ***key: value*** pairs. The `dict` used here is a dictionary where each key has another dictionary as it’s value. Each nested dictionary has a list as the value for each key.

</> **??** Line 7 in source **??** starts looping through file `f`. Line 8 checks if the current word `w` contains anything other than alphabetical characters and whether or not `w` is contained in the relevant stop-word file `lang` (for a list of English stopwords see appendix **??**). If both of those conditions are true, a variable `y` is created on line 9 (such as ‘l_00’ based on ‘00.faustroll.txt’) and `w` is added to the relevant dictionary file `dic` together with `y` and the current position `x` on line 10. After all files are processed, the two index structures look roughly like this:

§ **??**

```
{
  word1: {fileA: [pos1, pos2, ...], fileB: [pos], ...},
```

```python
# f = input text
# lang = stopwords
# dic = dictionary
# d = 'l' for Faustroll or 's' for Shakespeare
def setupcorpus(f, lang, dic, d):
  # x = counter, w = word in file f
  for x, w in enumerate(f):
    if w.isalpha() and (w.lower() not in lang):
      y = d + '_' + (re.search(r"((\d\d).(\w)+.txt)",
       ↪ f.fileid)).group(2)
      dic[w.lower()][y].append(x)
```

Code 1.2 – 'setupcorpus': processing a text file and adding to the index—Python

```
  word2: {fileC: [pos1, pos2], fileK: [pos], ...},
  ...
}
```

Using one of the terms from figure 1.2 on page 4 as an example, here are their entries in the index file (the files are represented by their number in the corpus, i.e. `l_00` is the 'Faustroll' file, `l_01` is the 'Poe' file, etc.). An excerpt from the actual `l_dict` can be found in the appendix **??**. 1.2

```
{
  doctor: {
    l_00: [253, 583, 604, 606, 644, 1318, 1471, 1858, 2334, 2431, 2446, 3039,
          ↪ 4743, 5034, 5107, 5437, 5824, 6195, 6228, 6955, 7305, 7822, 7892,
          ↪ 10049, 10629, 11055, 11457, 12059, 13978, 14570, 14850, 15063,
          ↪ 15099, 15259, 15959, 16193, 16561, 16610, 17866, 19184, 19501,
          ↪ 19631, 21806, 22570, 24867],
    l_01: [96659, 294479, 294556, 294648, 296748, 316773, 317841, 317854,
          ↪ 317928, 317990, 318461, 332118, 338470, 340548, 341252, 383921,
          ↪ 384136, 452830, 453015, 454044, 454160, 454421, 454596, 454712,
          ↪ 454796, 454846, 455030, 455278, 455760, 455874, 456023, 456123,
          ↪ 456188, 456481, 456796, 457106, 457653, 457714, 457823, 457894,
          ↪ 458571, 458918, 458998, 459654, 459771, 490749],
    l_02: [11476, 12098, 28151, 36270], ...
  }, ...
}
```

1.2 Text

After the setup stage is completed and the webpage is fully loaded, user input in the form of a text query is required to trigger the three pataphysical algorithms.

Image and video search do not use all three algorithms — where relevant this is highlighted in each section. Generally the following descriptions refer to the text search functionality only.

⧉ **??** Figure **??** previously showed the rough sequence of events in text search and highlighted that the pataphysicalisation from query to patadata happens in the `textsurfer.py` Python script file.

1.2.1 Clinamen

§ **??** The clinamen was introduced in chapter **??** but to briefly summarise it, it is the unpredictable swerve that Bök calls "the smallest possible aberration that can make the greatest possible difference" (**Bok2002**).

> Like all digitally encoded information, it has unavoidably the uncomfortable property that the smallest possible perturbations —i.e. changes of a single bit— can have the most drastic consequences. (**Dijkstra1988**)

In simple terms, the clinamen algorithm works in two steps:

1. get clinamen words based on dameraulevenshtein and faustroll text,
2. get sentences from corpus that match clinamen words.

</> **??** It uses the *Faustroll* (**Jarry1996**) as a base document and the Damerau-Levenshtein algorithm (**Damerau1964**; **Levenshtein1966**) (which measures the distance between two strings (with 0 indicating equality) to find words that are similar but not quite the same. The distance is calculated using insertion, deletion, substitution of a single character, or transposition of two adjacent characters. This means that we are basically forcing the program to return matches that are of distance two or one, meaning they have two or one spelling errors in them.

```python
# w = query word
# c = corpus
# i = assigned distance
def clinamen(w, c, i):
  # l_00 = Faustroll text
  words = set([term for term in l_00 if dameraulevenshtein(w, term) <=
   ↪  i])
  out, sources, total = get_results(words, 'Clinamen', c)
  return out, words, sources, total
```

Code 1.3 – 'clinamen': pataphysicalising a query term—Python

Source **??** line 6 creates the set of clinamen words using a list comprehension. It retrieves matches from the Faustroll file `l_00` with the condition that they are of Damerau-Levenshtein distance `i` or less to the query term `w` (see source **??**). Duplicates are removed. Line 7 then makes a call to the generic `get_results` function to get all relevant result sentences, the list of source files and the total number of results. </> **??** </> **??** </> **??**

```
# Michael Homer 2009
# MIT license
def dameraulevenshtein(seq1, seq2):
  oneago = None
  thisrow = range(1, len(seq2) + 1) + [0]
  for x in xrange(len(seq1)):
    twoago, oneago, thisrow = oneago, thisrow, [0] * len(seq2) + [x + 1]
    for y in xrange(len(seq2)):
      delcost = oneago[y] + 1
      addcost = thisrow[y - 1] + 1
      subcost = oneago[y - 1] + (seq1[x] != seq2[y])
      thisrow[y] = min(delcost, addcost, subcost)
      if (x > 0 and y > 0 and seq1[x] == seq2[y - 1] and
        seq1[x - 1] == seq2[y] and seq1[x] != seq2[y]):
          thisrow[y] = min(thisrow[y], twoago[y - 2] + 1)
  return thisrow[len(seq2) - 1]
```

Code 1.4 – Damerau-Levenshtein algorithm (**Homer2009**)—Python

The clinamen algorithm mimics the unpredictable swerve, the smallest possible aberration that can make the greatest possible difference, or the smallest possible perturbations with the most drastic consequences.

1.2.2 Result Sentences

The `get_results` function (see source **??**) is used by all three text algorithms (clinamen, syzygy and antinomy). Given the nested structure of the indexes `l_dict` and `s_dict`, the function loops through each of the `words` passed to it (`r`) first and then each file in `files.items()`. Lines 8 and 9 retrieve the dictionary of files for term `r` from the relevant dictionary. Line 13 gets the author and full title of file `e` and adds it to the list of sources in line 14. Line 15 makes use of another function called `pp_sent` (see source **??**) to get an actual sentence fragment for the current word `r` in file `e`, which is then added to the output. The output is structured as a triple containing the author and title, the list of resulting sentences and the name of the algorithm used. </> **??** </> **??**

In function `pp_sent` (source **??**) line 5 is important to note because it is a key functionality point. Even though the index files store a full list of all possible </> **??**

```python
# words = patadata words
# algo = name of algorithm
# corp = name of corpus
def get_results(words, algo, corp):
  total = 0
  out, sources = set(), set()
  for r in words:
    if corp == 'faustroll': files = l_dict[r]
    else: files = s_dict[r]
    # e = current file
    # p = list of positions for term r in file e
    for e, p in files.items():
      f = get_title(e)
      sources.add(f)
      o = (f, pp_sent(r.lower(), e, p), algo)
      total += 1
      out.add(o)
  return out, sources, total
```

Code 1.5 – ‘get_results’: retrieving all sentences for a list of words—Python

positions of a given word in each file, the `pp_sent` function only retrieves the sentence of the very first occurrence of the word rather than each one. This decision was taken to avoid overcrowding of results for the same keyword and is further discussed in chapter **??**. § **??**

Line 8 creates a list of punctuation marks needed to determine a suitable sentence fragment. Lines 9–17 and 18–26 set the `pos_b` (position before) and `pos_a` (position after) variables respectively. These positions can be up to 10 words before and after the keyword `w` depending on the sentence structure (punctuation marks). In line 28 the actual sentence fragment up to the keyword is retrieved, while in line 29 the fragment just after the keyword is retrieved. `ff[pos_b:pos]` for example returns the list of words from position `pos_b` to position `pos` from file `ff`. The built-in Python `.join()` function then concatenates these words into one long string separated by spaces. On line 30 a triple containing the pre-sentence, keyword and post-sentence is set as the output and then returned.

1.2.3 Syzygy

§ **??**

The concept of the syzygy was introduced in chapter **??** but can be roughly described as surprising and confusing. It originally comes from astronomy and denotes the alignment of three celestial bodies in a straight line. In a pataphysical context it is the pun. It usually describes a conjunction of things, something unexpected and surprising. Unlike serendipity, a simple chance encounter, the

```python
# w = the word (lower case)
# f = the file
# p = the list of positions
def pp_sent(w, f, p):
  out, pos = [], p[0] # FIRST OCCURRENCE
  ff = eval(f)
  pos_b, pos_a = pos, pos
  punct = [',', '.', '!', '?', '(', ')', ':', ';', '\n', '-', '_']
  for i in range(1, 10):
    if pos > i:
      if ff[pos - i] in punct:
        pos_b = pos - (i - 1)
        break
      else:
        if ff[pos - 5]: pos_b = pos - 5
        else:           pos_b = pos
    else: pos_b = pos
  for j in range(1, 10):
    if (pos + j) < len(ff):
      if ff[pos + j] in punct:
        pos_a = pos + j
        break
      else:
        if ff[pos + j]: pos_a = pos + j
        else:           pos_a = pos
    else: pos_a = pos
  if pos_b >= 0 and pos_a <= len(ff):
    pre = ' '.join(ff[pos_b:pos])
    post = ' '.join(ff[pos+1:pos_a])
    out = (pre, w, post)
  return out
```

Code 1.6 – 'pp_sent': retrieving one sentence—Python

syzygy has a more scientific purpose. In simple terms, the syzygy algorithm works in two steps:

1. get syzygy words based on synsets and hypo-, hyper-, holo- and meronyms from WordNet,
2. get sentences from corpus that match syzygy words.

The syzygy function makes heavy use of WordNet (**Miller1995**) through the **NLTK!** Python library (**NLTK2016**) to find suitable results (importing it using the following command `from nltk.corpus import wordnet as wn`). Specifically, as shown in source **??**, the algorithm fetches the set of synonyms (synsets) on line 5. It then loops through all individual items `ws` in the list of synonyms `wordsets` </> **??**

in line 7–20. It finds any hyponyms, hypernyms,holonyms, and meronyms for `ws` (each of which denotes some sort of relationship or membership with its parent synonym—see figure **??**) using the `get_nym` function (see lines 8, 11, 14, and 17). Line 21 makes use of the `get_results` function (see source **??**) in the same was as the clinamen function does.

▣ **??**
</> **??**
</> **??**

```
 1  # w = word
 2  # c = corpus
 3  def syzygy(w, c):
 4    words, hypos, hypers, holos, meros = set(),set(),set(),set(),set()
 5    wordsets = wn.synsets(w)
 6    hypo_len, hyper_len, holo_len, mero_len, syno_len = 0,0,0,0,0
 7    for ws in wordsets:
 8      hypos.update(get_nym('hypo', ws))
 9      hypo_len += len(hypos)
10      words.update(hypos)
11      hypers.update(get_nym('hyper', ws))
12      hyper_len += len(hypers)
13      words.update(hypers)
14      holos.update(get_nym('holo', ws))
15      holo_len += len(holos)
16      words.update(holos)
17      meros.update(get_nym('mero', ws))
18      mero_len += len(meros)
19      words.update(meros)
20      syno_len += 1
21    out, sources, total = get_results(words, 'Syzygy', c)
22    return out, words, sources, total
```

Code 1.7 – 'syzygy': pataphysicalising a query term—Python

</> **??**

The `get_nym` function in source **??** shows how the relevant 'nyms' are retrieved for a given synset. Line 5 initialises the variable `hhh` which gets overwritten later on. Several `if` statements separate out the code run for the different 'nyms'. Lines 6–7 retrieves any hyponyms using **NLTK!**'s `hyponyms()` function . Similarly lines 8–9 retrieve hypernyms, lines 10–14 retrieve holonyms, and lines 15–19 retrieve meronyms. Finally, line 20–23 adds the contents of `hhh` to the output of the function.

The syzygy algorithm mimics an alignment of three words in a line (query → synonym → hypo/hyper/holo/meronym).

1.2.4 Antinomy

§ **??**

The antimony, in a pataphysical sense, is the mutually incompatible. It was previously introduced in chapter **??**. In simple terms, the antinomy algorithm

```python
# nym = name of nym
# wset = synset
def get_nym(nym, wset):
  out = []
  hhh = wset.hyponyms()
  if nym == 'hypo':
    hhh = wset.hyponyms()
  if nym == 'hyper':
    hhh = wset.hypernyms()
  if nym == 'holo':
    hhhm = wset.member_holonyms()
    hhhs = wset.substance_holonyms()
    hhhp = wset.part_holonyms()
    hhh = hhhm + hhhs + hhhp
  if nym == 'mero':
    hhhm = wset.member_meronyms()
    hhhs = wset.substance_meronyms()
    hhhp = wset.part_meronyms()
    hhh = hhhm + hhhs + hhhp
  if len(hhh) > 0:
    for h in hhh:
      for l in h.lemmas():
        out.append(str(l.name()))
  return out
```

Code 1.8 – 'get_nym': retrieving hypo/hyper/holo/meronyms—Python

works in two steps:

1. get antinomy words based on synsets and antonyms from WordNet,
2. get sentences from corpus that match antinomy words.

</> **??**

For the antinomy I simply used WordNet's antonyms (opposites) (source **??**). In principle, this function is similar to the algorithm for the syzygy. It finds all antonyms through **NLTK!**'s `lemmas()[0].antonyms()` function on line 7 and retrieves result sentences using the `get_results` function on line 12.

</> **??**

The antinomy algorithm mimics the mutually incompatible or polar opposites.

1.2.5 Formalisation

§ 1.1.1

A formal description of the `pata.physics.wtf` system in terms of an **IR!** (**IR!**) model described in chapter 1.1.1 is unsuitable. It assumes for example the presence of some sort of ranking algorithm $R(q_i, d_j)$.

```python
# w = input query term
# c = name of corpus
def antinomy(w, c):
  words = set()
  wordsets = wn.synsets(w)
  for ws in wordsets:
    anti = ws.lemmas()[0].antonyms()
    if len(anti) > 0:
      for a in anti:
        if str(a.name()) != w:
          words.add(str(a.name()))
  out, sources, total = get_results(words, 'Antinomy', c)
  return out, words, sources, total
```

Code 1.9 – 'antinomy': pataphysicalising a query term—Python

Making relevant changes (e.g. exchanging the ranking function for a pataphysicalisation function) to the specification by Baeza-Yates and Ribeiro-Neto (**Baeza-Yates2011**), an approximate system description for the Faustroll corpus text search could be as follows.

D	$=$ the set of documents $\{d_1, \ldots, d_m\}$
m	$=$ the number of all documents in D ($|D| = 28$)
V	$=$ the set of all distinct terms $\{v_1, \ldots, v_n\}$ in D not including stopwords
q	$=$ the user query
F	$=$ the set of patalgorithms $\{f_C, f_S, f_A\}$
P	$=$ the set of pataphysicalised query terms $\{p_1, \ldots, p_u\}$
u	$=$ the number of terms in P
$P(q)$	$=$ the set of patadata $\{P(q)_C \cup P(q)_S \cup P(q)_A\}$ for query q
R	$=$ the set of results $\{r_1, \ldots, r_o\}$
o	$=$ the number of results in R
$R(P(q))$	$=$ the set of results $\{R(P(q)_C) \cup R(P(q)_S) \cup R(P(q)_A)\}$ produced by each algorithm in F
r	$=$ a result of form (d, sentence, f)

We can then define the three patalgorithms in a more formal way as shown in equations **??**, **??**, and **??**.

$$P(q)_C = \{p \in v_0 : 0 < \text{dameraulevenshtein}(q, p) \leq 2\} \tag{1.1}$$

Σ ??
</> ??

`damerauleveshtein(q,p)` in equation **??** is the Damerau-Levenshtein algorithm as described in section **??** and v_0 is the Faustroll text.

$$\begin{aligned} P(q)_S &= \{p \in V : p \in \text{nyms}(s),\ \forall s \in \text{synonyms}(q)\} \\ \text{where } \text{nyms}(s) &= \text{hypos}(s)\ \cup\ \text{hypers}(s)\ \cup\ \text{holos}(s)\ \cup\ \text{meros}(s) \end{aligned} \tag{1.2}$$

`synonyms(q)` in equation **??** is the WordNet/**NLTK!** function to retrieve all synsets for the query q and the four 'nym' functions return the relevant hyponyms, hypernyms, holonyms or meronyms for each of the synonyms. Σ **??**

$$P(q)_A = \{p \in V : p \in \text{antonyms}(s),\ \forall s \in \text{synonyms}(q)\} \tag{1.3}$$

Similarly, in equation **??** the `synonyms(q)` function returns WordNet synsets for the query q and the `antonyms(s)` function returns WordNet antonyms for each of the synonyms. Σ **??**

$$R(P(q)) = \{(d \in D,\ \mathit{sent}(p) \in d,\ f \in F) : \forall\ p \in P(q)_f))\} \tag{1.4}$$

The set of results $R(P(q))$ can then be defined as shown in equation **??**. It returns a list of triples containing the source text d, the sentence `sent(p)` and the algorithm f. For each pataphysicalised query term p one sentence is retrieved per file d. Σ **??**

1.3 Image & Video

The image and video search of `pata.physics.wtf` both work slightly differently to the text search described in section **??**. In simple terms, the image and video search works in three steps: § **??**

1. translate query,
2. pataphysicalise the translation,
3. retrieve matching images/videos using **API!** calls.

</> **??**

The first step is to translate the search terms as shown in source **??**. Lines 2 and 4 set up the **API!** connection to the Microsoft Translator tool (**TranslatorAPI**) given an ID and 'secret', neither of which are included here for security reasons. The query `sent` then passes through a chain (alignment) of three translations in true syzygy fashion: from English → French, from French → Japanese, and from Japanese → English (lines 5–7). All three languages are then returned in a triple (line 8).

```python
# sent = the query string
from microsofttranslator import Translator
def transent(sent):
  translator = Translator(microsoft_id, microsoft_secret)
  french = translator.translate(sent, "fr")
  japanese = translator.translate(french, "ja")
  patawords = translator.translate(japanese, "en")
  translations = (french, japanese, patawords)
  return translations
```

Code 1.10 – `transent': translating query between English-French-Japanese-English—Python

The next step is to pataphysicalise the translated query (see source **??**). The `pataphysicalise` function transforms this translation in a process slighlty sim- </> **??** plified from the `syzygy` algorithm. The decision to simplify the algorithm was made due to performance issues related to the **API!** calls that follow in the final step of the search process.

In line 5 WordNet synsets are retrieved using **NLTK!**'s `synsets` function. For each of these synsets we get a list of synonyms (line 8) which we add to the output in a normalised form (line 11) removing any underscores if there are any.

```python
 1  # words = query term(s)
 2  def pataphysicalise(words):
 3    sys_ws = set()
 4    for word in words:
 5      synonyms = wn.synsets(word)
 6      if len(synonyms) > 0:
 7        for s in synonyms:
 8          for l in s.lemmas():
 9            x = str(l.name())
10            o = x.replace('_', ' ')
11            sys_ws.add(o)
12    return sys_ws
```

Code 1.11 – `pataphysicalise': pataphysicalise image and video query terms—Python

▣ **??** Figure **??** previously showed the rough sequence of events in an image and video search and highlighted that the pataphysicalisation from query to patadata happens in the `imgsurfer.py` Python script file while the production of results from that patadata happens in the `fania.js` JavaScript file.

And finally, **API!** calls to the various external tools are made. This is described § **??** in section **??** below.

1.3.1 REST & API

The final step of the image and video search process described on page **??** is to retrieve matching images/videos using **API!** calls to Flickr (**FlickrAPI**; **FlickrGuideAPI**), Getty (**GettyAPI**; **GettyOverviewAPI**), Bing (**BingAPI**; **BingAzureAPI**), YouTube (**YouTubeAPI**) and Microsoft Translator (**TranslatorAPI**).

The patadata used to make the **API!** calls is limited to 10 keywords and uses the function `random.sample(pata, 10)`, where `pata` is the set of terms obtained by pataphysicalising the query translation.

A **REST!**ful **API!** allows browsers ('clients') to communicate with a web server via **HTTP!** methods such as GET and POST. The idea is that a given service, like the Microsoft Bing search **API!**, can be accessed in a few simple steps using **JSON!** (**JSON!**) (**JSON2016**). These are:

1. for each of the 10 query terms do:
   a) construct the **URL!** (**URL!**) with the query request
   b) setup authentication
   c) send **URL!** and authentication
   d) receive response in **JSON!**
   e) add result to output list `imglist`
2. once 10 results are reached, render results as spiral

Source **??** shows how such an **API!** call is made using JavaScript (in this case Flickr). Source **??** below shows how 10 seperate images are collected into one results list and the `createSpiral` function is called to render the images to the user in **HTML!** (see appendix **??** for the relevant code snippet).

</> **??**
</> **??**
§ **??**

The Bing and Getty searches work in a similar way with one exception. Getty does not populate the output list by doing 10 individual **API!** calls but rather by adding 10 results from 1 call. This is due to a time restriction in the Getty **API!**; it doesn not allow 10 calls in a second.

൭ ൭ ൭

An example **URL!** request for the Flickr image search with the query term of 'kittens' and a requested response format of **JSON!** is this: `http://api.flickr.com/services/feeds/photos_public.gne?jsoncallback=?tags=kittens&tagmode=all&format=json`. Flickr will then send back the response in **JSON!**

```javascript
function flickrsearch(patadata){
  for(var x=0; x<10; x++){
    $.getJSON("http://api.flickr.com/services/feeds/photos_public.gne
    ↪  ?jsoncallback=?",
      {
        tags: patadata[x].query,
        tagmode: "all",
        format: "json"
      },
      function(data,status,ajax) {
        var title = "", media = "", link = "";
        if (data.items[0] != undefined) {
          title = data.items[0].title;
          media = data.items[0].media.m;
          link = data.items[0].link;
        }
        imgList([title, media, link]);
      }
    );
  }
};
```

Code 1.12 – ‘flickrsearch’: using the Flickr API to retrieve images—JavaScript

```javascript
var allImages = [];
function imgList(img){
  if (allImages[0] != "") {
    allImages.push(img);
  }
  if (allImages.length === 10) {
    createSpiral(allImages);
  }
}
```

Code 1.13 – ‘imgList’: accumulates 10 images and calls the ‘createSpiral’ function—JavaScript format. One entry of the list of results is shown below (with whitespace formatting added for convenience). The algorithm in source **??** only retrieves the `data.items[0].title`, `data.items[0].media.m` and `data.items[0].link` (lines 12, 13, and 14) and ignores all other data fields. </> **??**

```
({...
  "items":
    [{
      "title": "P_20161101_191123",
      "link": "http://www.flickr.com/photos/pinknancy/30078720153/",
      "media": {"m":"http://farm6.staticflickr.com/5759/30078720153_f03e036e89⌋
            ↪  _m.jpg"},
      "date_taken": "2016-11-01T19:11:23-08:00",
      "description": ...,
      "published": "2016-11-01T15:28:10Z",
      "author": "nobody@flickr.com (pinknancy)",
      "author_id": "8748781@N08",
      "tags": ""
    },...]
})
```

Once the `imglist` contains 10 items it is passed to the `createSpiral` function which renders it to **HTML!**. Appendix **??** shows an example shortnened **JSON!** result from Bing. § **??**

๑ ๑ ๑

The video search also uses an **API!** to retrieve results. This function is written in Python and uses the *Requests* library (**Reitznd**) to make the **API!** calls to YouTube (**YouTubeAPI**) as shown in source **??**. </> **??**

</> **??**

First, the query is translated using the `transent` function on line 3. Line 4 seperates the English translation into its own list `transplit` which is then pataphysicalised on line 5 using the algorithm described in source **??**. </> **??**

Lines 6–9 construct the first part of the **URL!** to use for the **REST!** (**REST!**) request. Lines 10–23 then loop through each of the patadata terms generated by the `pataphysicalise` function on line 5 to make a call and retrieve some video details (title, thumbnail and ID) as seen on lines 17–19. On line 20 these details are added to the output list.

The video results are then also displayed in a golden spiral in the same way as the images. This is described in section **??**. § **??**

```python
def getvideos(query):
  out = []
  translations = transent(query)
  transplit = translations[2].split(' ')
  tmp = pataphysicalise(transplit)
  b0 = "https://www.googleapis.com/youtube/v3/search?"
  b1 = "&order=viewCount&part=snippet&"
  b3 = "&type=video&key=%s" % yt_key
  b4 = "&maxResults=10&safeSearch=strict"
  for x in tmp:
    y = ' '.join(x)
    b2 = "q=%s" % translations[2]
    yturl = ''.join([b0, b1, b2, b3, b4])
    vids = requests.get(yturl)
    if vids.json()['items']:
      for i in vids.json()['items']:
        vidtitle = i['snippet']['title']
        vidthumb = i['snippet']['thumbnails']['default']['url']
        vidid = i['id']['videoId']
        out.append((vidtitle, vidthumb, vidid))
      break
    else:
      out = []
  return out, translations
```

Code 1.14 – `getvideos': using the YouTube API to retrieve images—Python

1.4 Design

Once the patalgorithms have produced their respective results, the page displaying these results can be rendered. This is done using the templating language Jinja (**Jinja2016**) and **HTML!** (with **CSS!** stylesheets and some JavaScript).

§ **??** One of the key requirements for the *Syzygy Surfer* tool was that "the user should be able to choose the techniques they use" (**Hendler2011**). This has been adopted for pata.physics.wtf in the sense that the user has different options for the display of results.

The text results page has three different result styles, with 'Poetry — Queneau' being the default.

Poetry Displayed in sonnet style (two quatrains and two tercets) if possible, although no rhyming pattern is used[6].

- Queneau — Each line can be changed manually.

[6]This is addressed in chapter **??**.

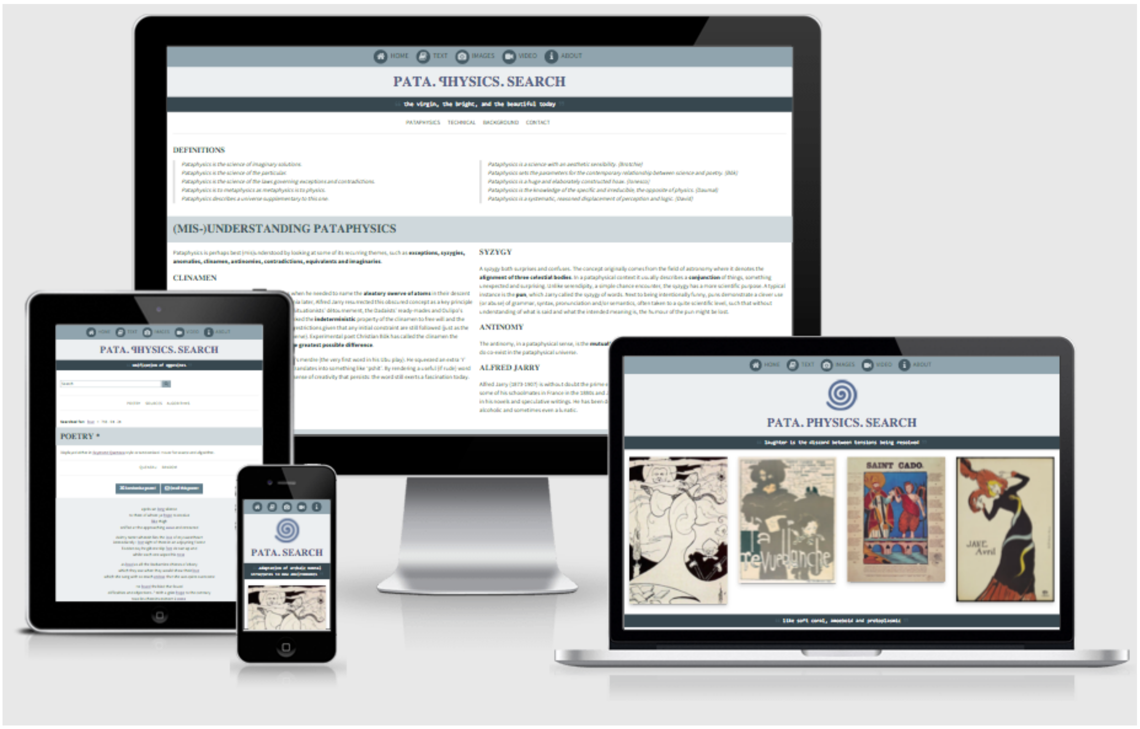


Figure 1.6 – Responsive design of pata.physics.wtf

- Random — The whole poem can be randomised.

Sources	Ordered by source text.
Algorithms	Ordered by algorithm.

The image and video results pages work the same way. They both have two display options, with the 'Spiral' option being the default. The spirals are modelled on the idea of golden spirals (more precisely an approximation in the form of a Fibonacci spiral).

Spiral	Displayed as square images/videos in a spiral.
List	Displayed as a simple list.

The overal visual design is shown in image **??**. **??**

1.4.1 Poetry

Source **??** shows the segment of **HTML!**/Jinja code that renders the Queneau poetry. The code renders the 4 stanzas of the poem. This is done using two nested Jinja `for` loops (line 2 and line 10). Line 2 loops through the (ideally) 14 lines of the poem. `lol` can be considered a masterlist of all sublists for each poem line. **??**

Functionality for sending the currently showing poem per email is added via a button which calls a JavaScript function `onclick="return getContent(this)"` which then retrieves the content of each line in the poem and sends it to the body of the email.

`all_sens` is the pool of all sentences. It is structured as follows.

```
[(title, (pre, word, post), algorithm), ...]
```

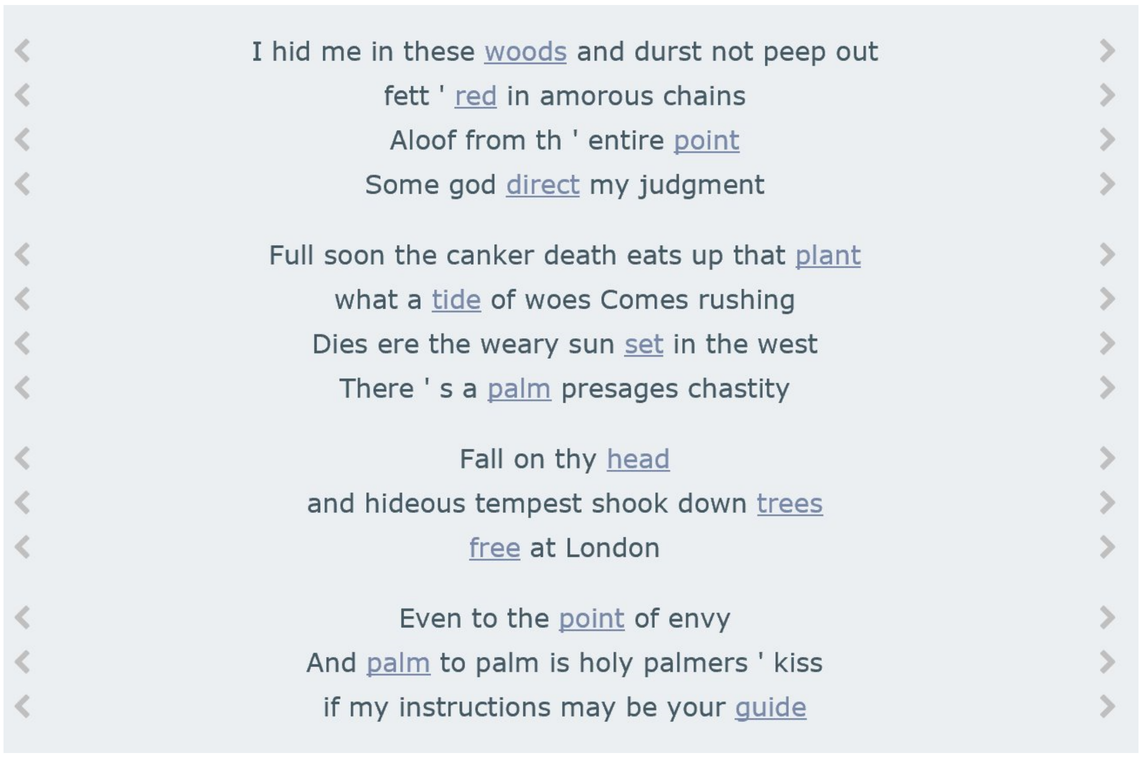
`lol` is a list subdivided into partitions for each line of the sonnet. Let's say there are 350 sentences overall in `all_sens`. To divide them equally among the 14 lines of a sonnet, we need to create `lol` with 14 equal parts of 25 sentences.

```
[all_sens[0-24], all_sens[25-49], ...,  all_sens[325-349]]
```

```
<div>
  {% for n in range(1, lol|length + 1) %}
    {% set wid = ["wn", n|string]|join %}
    {% set lid = ["lyr", n|string]|join %}
    {% set sid = ["scrollLinks", n|string]|join %}
    {% set aid = lol[n-1] %}
    <div id="poems">
      <div id="{{wid}}" class="wn">
        <div id="{{lid}}" class="lyr">
          {% for sens in aid %}<span title="{{ sens[0] }}, {{ sens[2]
           ↪  }}">{{ sens[1][0] }} <form class="inform"
           ↪  action="../textresults" method="post"><input
           ↪  class="inlink" type="submit" name="query" value="{{
           ↪  sens[1][1] }}" onclick="loading();"></input></form> {{
           ↪  sens[1][2] }}</span>{% endfor %}
        </div>
      </div>
      <div id="{{sid}}" class="scrollLinks"></div>
    </div>
  {% endfor %}
</div>
```

Code 1.15 – Simplified **HTML!** code for rendering Queneau style poems

?? Changing a line of the poem is achieved by clicking on one of the buttons on either side of the poem's line (as shown in image **??**). This will trigger a JavaScript function (based on (**DYNWEB2016**)) to automatically scroll to the next sentence.


I hid me in these woods and durst not peep out
fett ' red in amorous chains
Aloof from th ' entire point
Some god direct my judgment

Full soon the canker death eats up that plant
what a tide of woes Comes rushing
Dies ere the weary sun set in the west
There ' s a palm presages chastity

Fall on thy head
and hideous tempest shook down trees
free at London

Even to the point of envy
And palm to palm is holy palmers ' kiss
if my instructions may be your guide

Figure 1.7 – Example Queneau poem for query 'tree'

Non-Queneau poems have a slightly different functionality. It is not possible to change the poem line by line but rather the whole poem can be randomised on demand. This relies on a random number generator in JavaScript. A function `shufflePoem()` creates a random variable `r` as `Math.floor(Math.random() * n)`, which can then be used to generate a new list of 14 lines for the poem randomly selected from the pool of sentences `all_sens`.

1.4.2 Lists

The two other ways to display text results are as a list ordered by source or by patalgorithm which works in a similar way to what is described in source **??**. The code is wrapped in an **HTML!** unordered list tag `<ul>`. A Jinja `for` loop generates the individual `<li>` tags on line 4. </> **??**

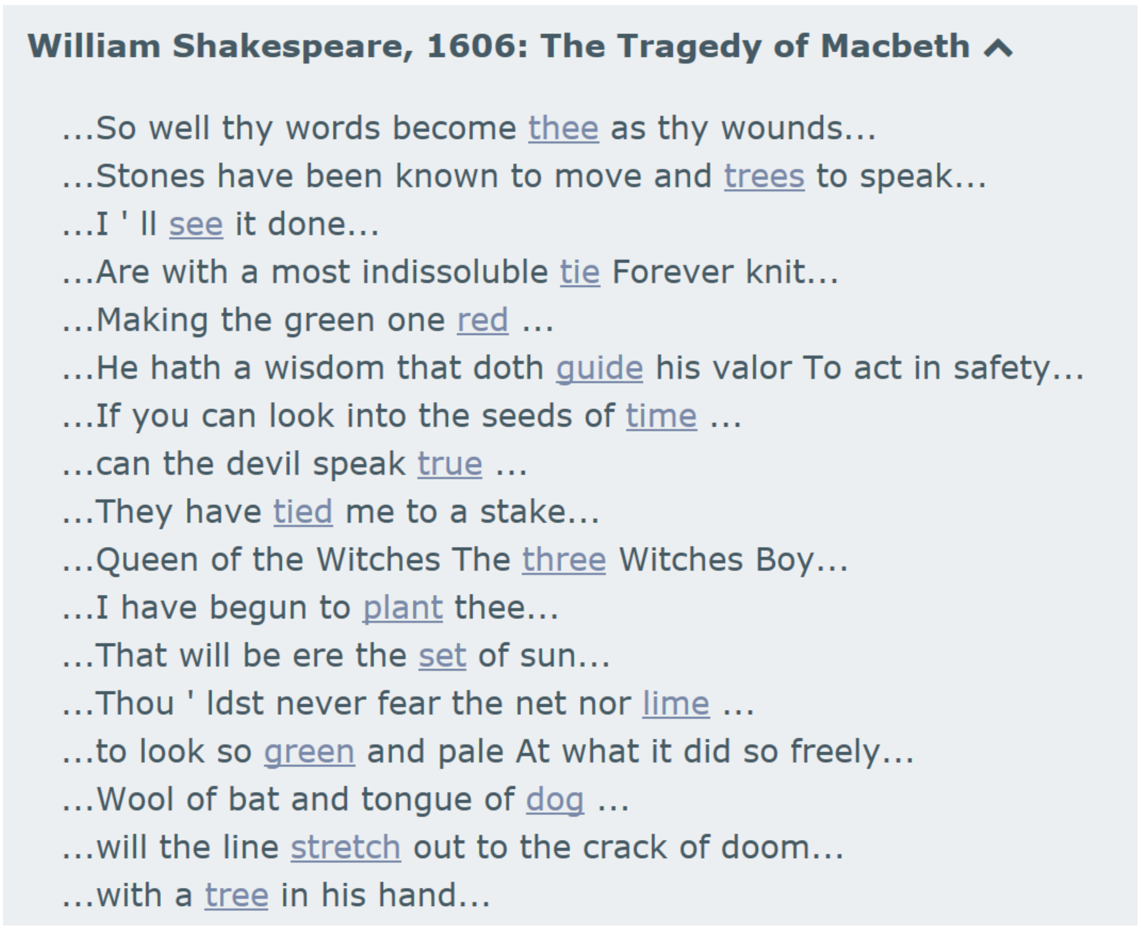
A `sens` in `all_sens` is structured as `(title, (pre, word, post), algorithm)`. This means that to access the name of the algorithm we need to call the Jinja template `{{ sens[2] }}`, to get the first half of the sentence we need `{{ sens[1][0] }}`, the middle keyword (i.e. the patadata term) `{{ sens[1][1] }}` and the second half of the sentence `{{ sens[1][2] }}`.

??

Image **??** shows a shortened example set of results for query 'tree' ordered by source, that is, ordered by original file.

```
<ul>
  {% for sens in all_sens %}
    {% if file == sens[0] %}
      <li title=`{{ sens[2] }}'>...{{ sens[1][0] }} <form class=`inform'
        ↪  action=`../textresults' method=`post'><input class=`w3-hide'
        ↪  type=`radio' name=`corpus' value=`{{ corpus }}'
        ↪  checked><input class=`inlink' type=`submit' name=`query'
        ↪  value=`{{ sens[1][1] }}' onclick=`loading();'></input></form>
        ↪  {{ sens[1][2] }}...</li>
    {% endif %}
  {% endfor %}
</ul>
```

Code 1.16 – Simplified **HTML!** code for rendering a list of text results by source

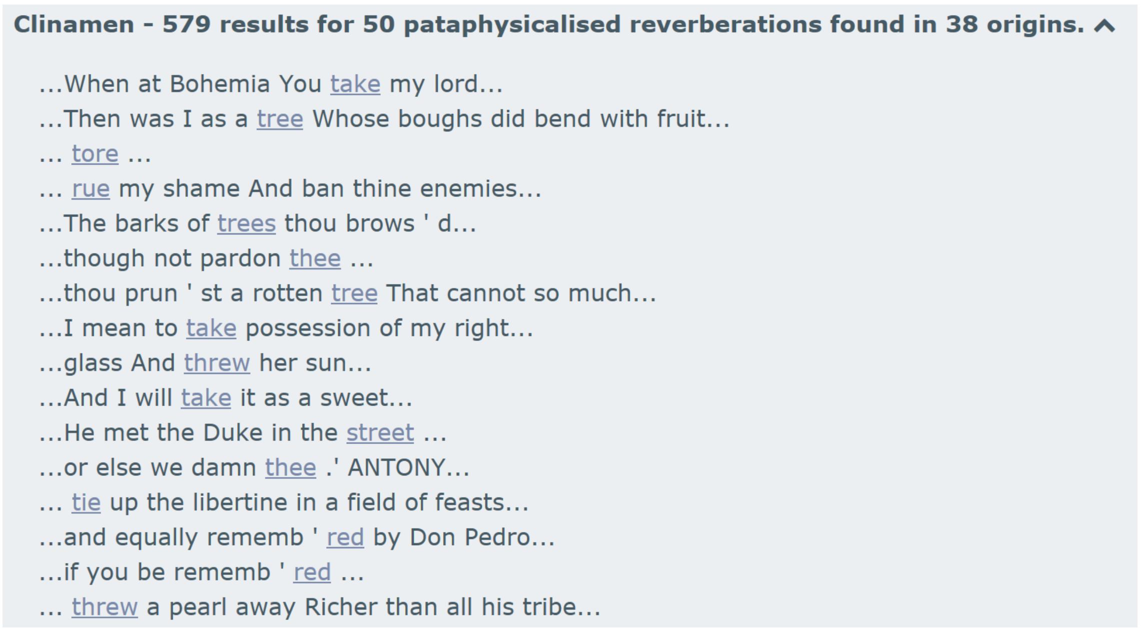


William Shakespeare, 1606: The Tragedy of Macbeth

...So well thy words become thee as thy wounds...
...Stones have been known to move and trees to speak...
...I ' ll see it done...
...Are with a most indissoluble tie Forever knit...
...Making the green one red ...
...He hath a wisdom that doth guide his valor To act in safety...
...If you can look into the seeds of time ...
...can the devil speak true ...
...They have tied me to a stake...
...Queen of the Witches The three Witches Boy...
...I have begun to plant thee...
...That will be ere the set of sun...
...Thou ' ldst never fear the net nor lime ...
...to look so green and pale At what it did so freely...
...Wool of bat and tongue of dog ...
...will the line stretch out to the crack of doom...
...with a tree in his hand...

Figure 1.8 – Example results for query ‘tree’ ordered by source

Image **??** shows a shortened example set of results for query 'tree' ordered by patalgorithm, that is, ordered by the algorithm which produced the patadata. **??**


Clinamen - 579 results for 50 pataphysicalised reverberations found in 38 origins.

...When at Bohemia You take my lord...
...Then was I as a tree Whose boughs did bend with fruit...
... tore ...
... rue my shame And ban thine enemies...
...The barks of trees thou brows ' d...
...though not pardon thee ...
...thou prun ' st a rotten tree That cannot so much...
...I mean to take possession of my right...
...glass And threw her sun...
...And I will take it as a sweet...
...He met the Duke in the street ...
...or else we damn thee .' ANTONY...
... tie up the libertine in a field of feasts...
...and equally rememb ' red by Don Pedro...
...if you be rememb ' red ...
... threw a pearl away Richer than all his tribe...

Figure 1.9 – Example results for query 'tree' ordered by patalgorithm

1.4.3 Spiral

The image and video spirals are constructed in complicated nested **HTML!** components. The code for generating an image spiral is shown in appendix **??**. The video spiral is constructed in a similar way but directly in the **HTML!** file as opposed to in the JavaScript file. The video spiral is almost identical, the only difference is that only the biggest 5 videos are atually embedded as videos. The smaller 5 videos are shown as still images which link to the relevant YouTube page. § **??**

Generally, the idea was taken from the pataphysical ***grand gidouille*** (see chapter **??**) and represented as a Fibonacci spiral. § **??**

Figure **??** shows a spiral created using the Flickr image search for query 'blue mountains' overlaid with a white Fibonacci spiral to highlight the structure. **??**

1.5 Prototypes

The final website `pata.physics.wtf` went through several iterations of development since it was first conceived in 2012. This included 3 major technical updates since the first prototype and 2 new visual re-designs.

Table **??** shows the main differences and similarities between the versions. **??**

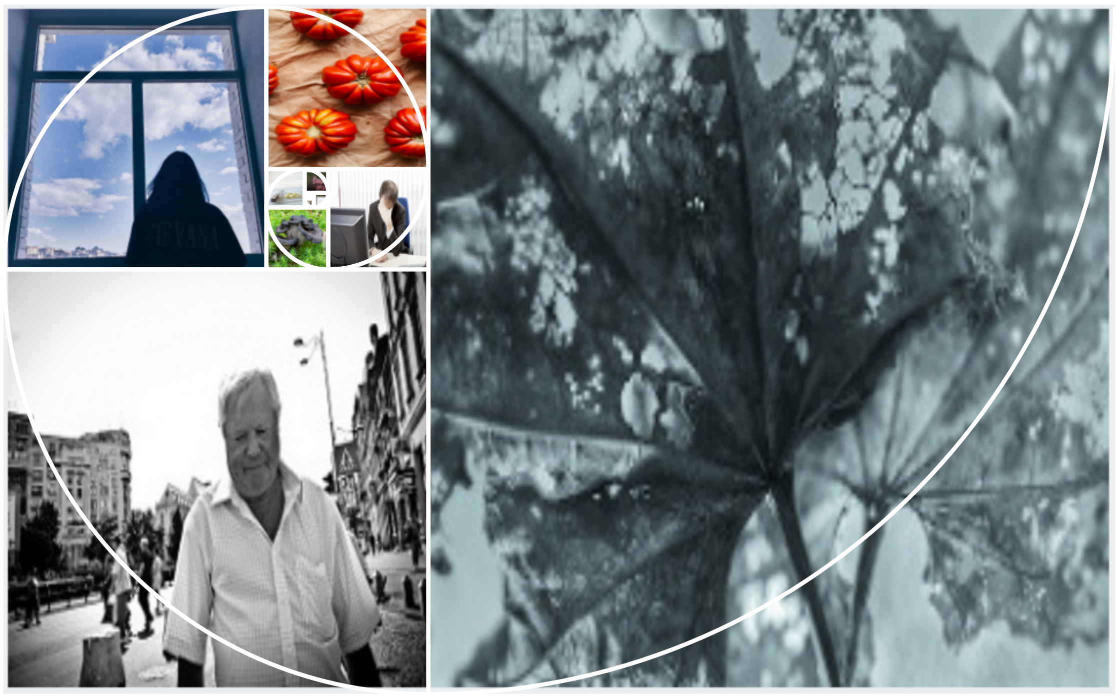


Figure 1.10 – Fibonacci spiral overlaid onto an image results for query 'blue mountains' using Flickr

Table 1.1 – Comparison of different versions of `pata.physics.wtf`

	Version 1	**Version 2**	**Version 3**	**Version 4**
Language(s)	Python, Django	Python, Flask	Python, Flask	Python, Flask, JavaScript
Server	Django, Heroku	Flask, Mnemosyne	Flask, Gunicorn, Mnemosyne	Flask, Gunicorn, OVH
Features	Text	Text, Image, Video	Text, Image, Video	Text, Image, Video
Corpus	Faustroll text	Faustroll text	Faustroll's library	Faustroll's library and Shakespeare
API's	WordNet	WordNet, Flickr, Bing, YouTube, Microsoft Translator	WordNet, Bing, YouTube, Microsoft Translator	WordNet, Flickr, Getty, Bing, YouTube, Microsoft Translator
Design	Algorithms	Algorithms, Spiral	Algorithms, Source, Poetry, Spiral, List	Algorithms, Source, Poetry, Spiral, List
Responsive	No	Yes	Yes	Yes

Images **??**, **??** and **??** show the 3 main visual designs.

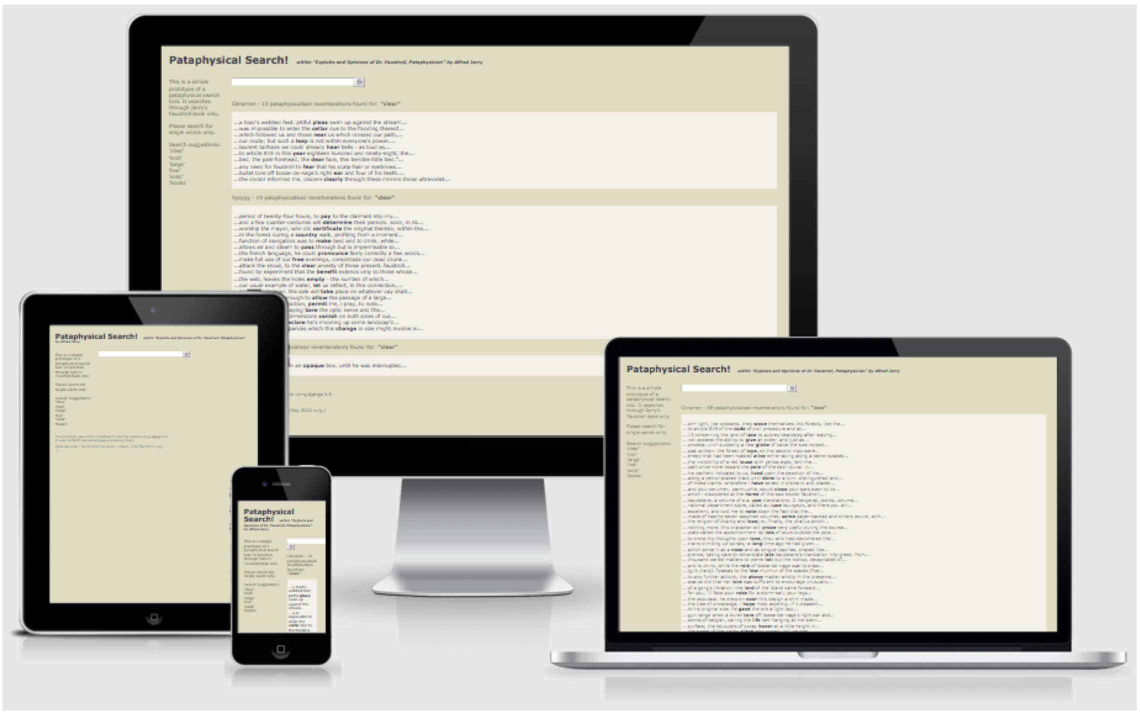


Figure 1.11 – First version of pata.physics.wtf

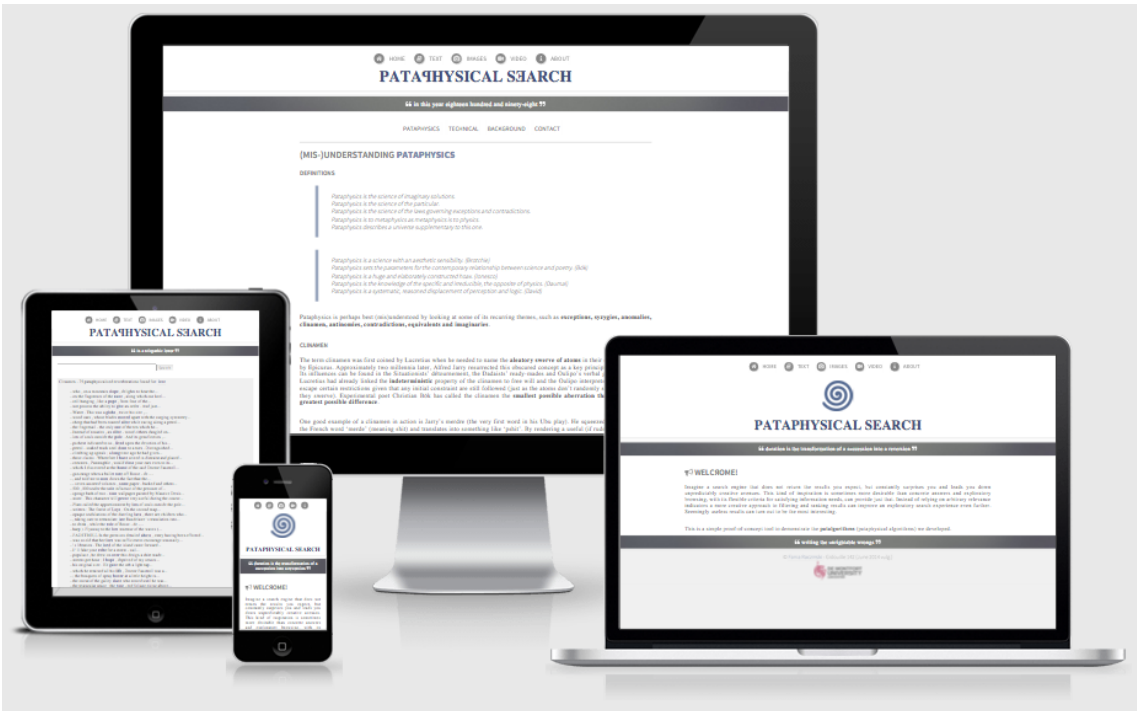


Figure 1.12 – Second major version of pata.physics.wtf

The latest version, which is now live at pata.physics.wtf, introduced major changes to the initial setup stage of the system and a lot of the code was refact-

ored and improved. As of the date of writing this, there were over 360 commits
§ **??** in the git repository since 2012. See appendix **??**.