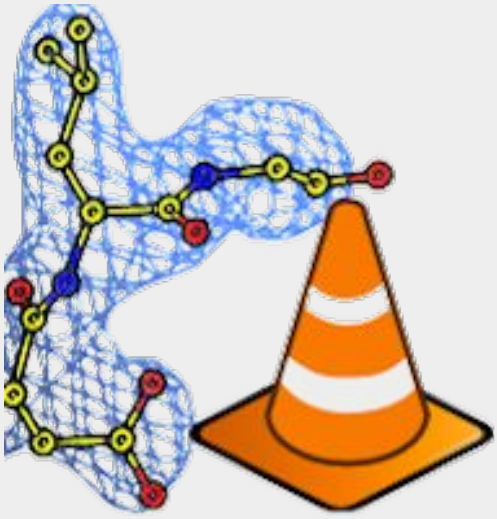
Model Building
Buccaneer & ModelCraft

Paul Bond
University of York

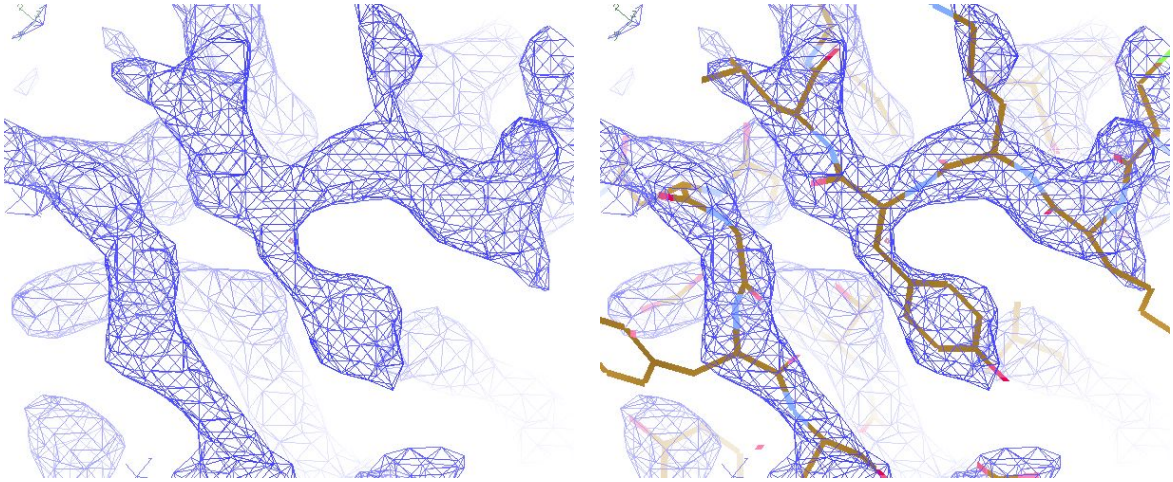
This presentation is on the automated model-building methods developed in the group of Kevin Cowtan at the University of York. Kevin wrote *Buccaneer* for building proteins and *Nautilus* for building nucleic acids, and *ModelCraft* is a new pipeline that includes both of those programs.

Buccaneer

Protein



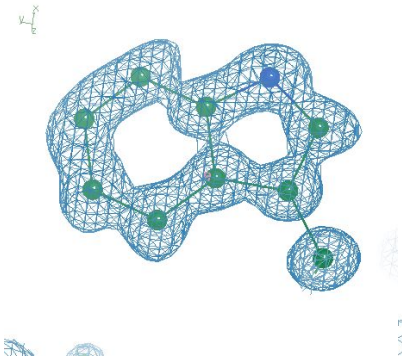
Buccaneer - Task



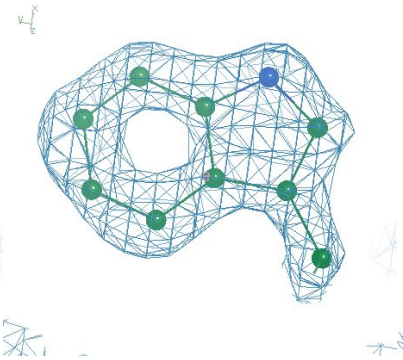
This is the general task of model building - taking a map and building an atomic model into it. At this point you have some initial phases either through experimental phasing or molecular replacement (MR), which might have been improved through density modification. If you need to build a model from scratch or improve the existing model it's usually better to start by trying an automated (i.e. non-interactive) model-building program. If your MR model is very good you can probably skip this step and go straight to interactive rebuilding and validation.

Buccaneer - Task

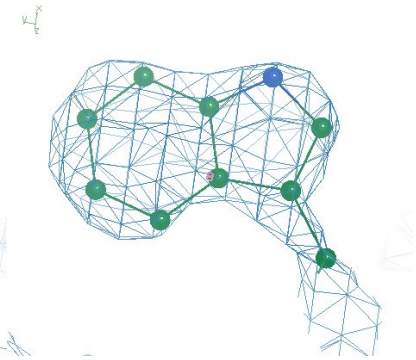
Resolution: 1Å



Resolution: 2Å



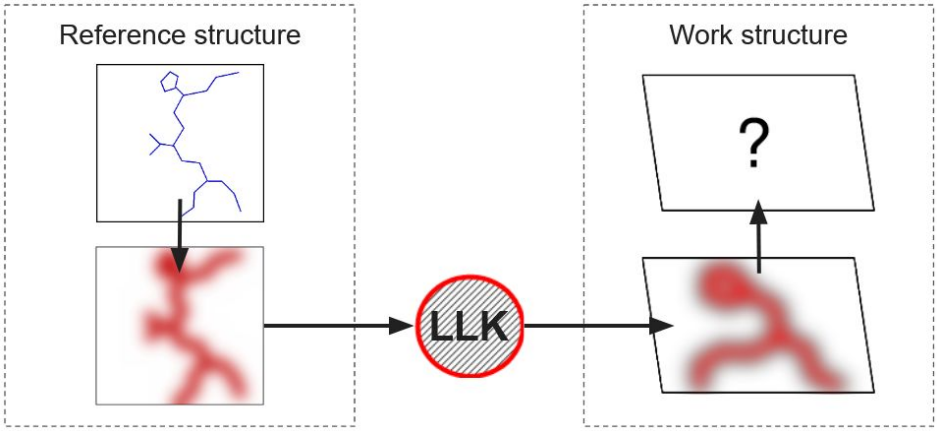
Resolution: 3Å



The *Buccaneer* method was designed to address the problem that electron density maps look very different at different resolutions. At high resolution you can distinguish peaks for individual atoms, but if you write a program that looks for separate atomic peaks then it wont work when you have a low resolution map and you can't see them.

Buccaneer - LLK target function

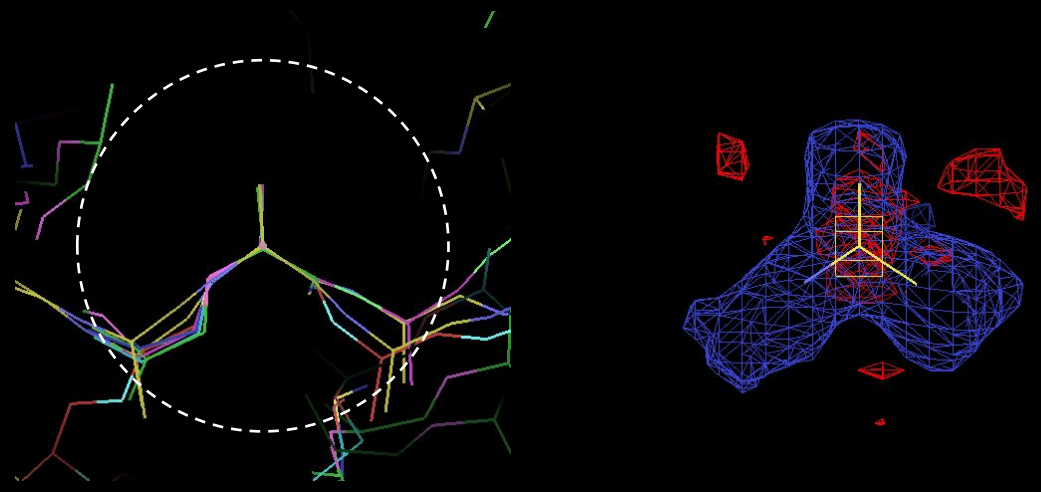
Compare simulated map and reference model to obtain likelihood target then search for this target in the unknown map



To help with this problem *Buccaneer* uses a known reference structure. You give B*uccaneer* the map you're trying to build a model into (the work map) and it generates a map for a reference structure with the same resolution and same level of noise as this map. Then it uses the reference structure to find out what features in the reference map look like, and looks for the same features in the work map.

Buccaneer - LLK target function

Measure mean and variance of reference map in a 4Å sphere around Cα



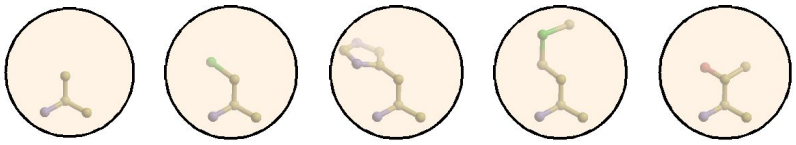
In practice, the target to search for individual residues is obtained by superimposing all the residues in the reference structure on top of each other. The mean and variance of the density is then used to construct a likelihood target within a 4A sphere. The blue density shows regions of conserved high density and the red density shows regions of conserved low density. Both are important when finding residues in the work map. If there is low density in the blue regions or high density in the red regions then it is likely not the correct position for a residue.

Buccaneer - LLK target function

Likelihood functions based on conserved density features

Finding, Growing Cα environment (4.0Å sphere)

Sequencing Cβ environment (5.5Å sphere)



ALA CYS HIS MET THR … x20

The 4A C-alpha target is used in the first step of the program to find initial residue positions and the second step to grow those residues into chains. The program also uses 5.5A targets centred around C-beta atoms. A different target is made for each residue type and those are used for sequencing. Rotamers are not separated before making the targets, so the mean and variance of the density is calculated for side chains pointing in lots of different directions, but this still provides enough information to find the correct sequence.

Buccaneer - Steps

1. **Find** oriented residue positions
2. **Grow** them into chain fragments
3. **Join** overlapping fragments and resolve branches
4. **Link** nearby termini
5. **Sequence** the chains
6. **Correct** insertions and deletions
7. **Filter** chains in poor density
8. **NCS** superposition to extend chains
9. **Prune** residues to resolve clashes
10. **Rebuild** side chains

Buccaneer runs as a cyclic calculation, where each cycle has these 10 steps.

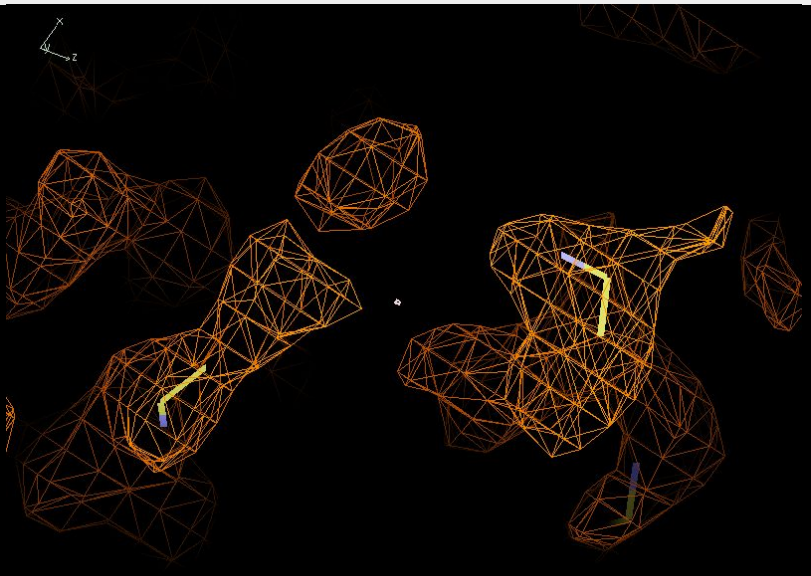
Buccaneer - Example

Case Study

A difficult loop in a 2.9Å map
calculated using real data from the JCSG

Buccaneer - Example

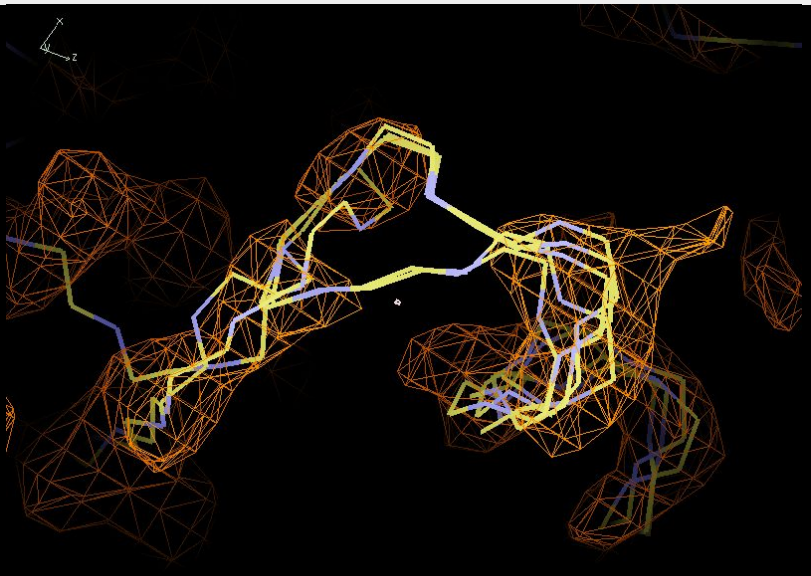
1. ***Find***
2. Grow
3. Join
4. Link
5. Sequence
6. Correct
7. Filter
8. NCS
9. Prune
10. Rebuild



The first step is to find possible positions for new residues. It has placed residues at three positions in this figure (two in the foreground and one in the background) where the density has the right shape according to the C-alpha target function. There are also other residues placed out of this view.

Buccaneer - Example

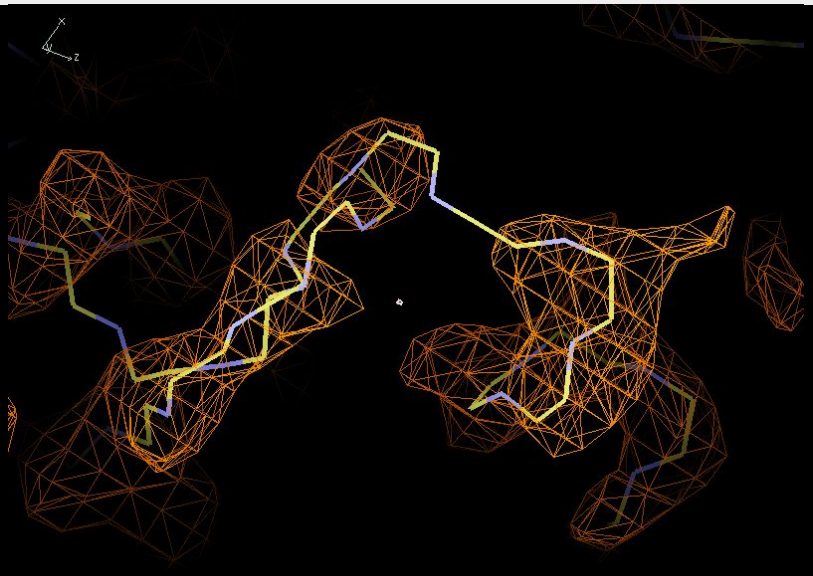
1. Find
2. ***Grow***
3. Join
4. Link
5. Sequence
6. Correct
7. Filter
8. NCS
9. Prune
10. Rebuild



The next step is to grow each residue into a long chain fragment. New residues get added at both ends using an exhaustive search over allowed Ramachandran angles and the same C-alpha target. Once the target score drops below a threshold value the growing stops. This step gives lots of fragments, many of which overlap. In this example it has built a helix on the right where the chains agree fairly well, two different paths to bridge the gap in the middle and a contradictory chain on the left that has been built in the opposite direction.

Buccaneer - Example

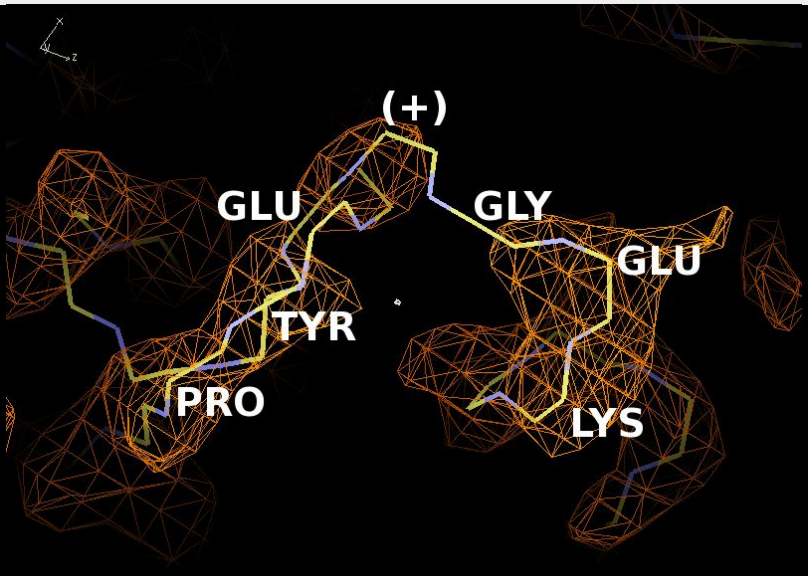
1. Find
2. Grow
3. ***Join***
4. Link
5. Sequence
6. Correct
7. Filter
8. NCS
9. Prune
10. Rebuild



The joining step merges all the chain fragments that agree with each other into single chains. This step also has to make decisions about which routes to follow. In addition to looking at how well the residues fit the density, it also tries to build the longest chains (e.g. it chose the longest route over the middle loop). This bias towards long chains helps to build helices correctly. The chain built in the reversed direction is still there because it can't be merged and at this stage it hasn't been decided which is correct.

Buccaneer - Example

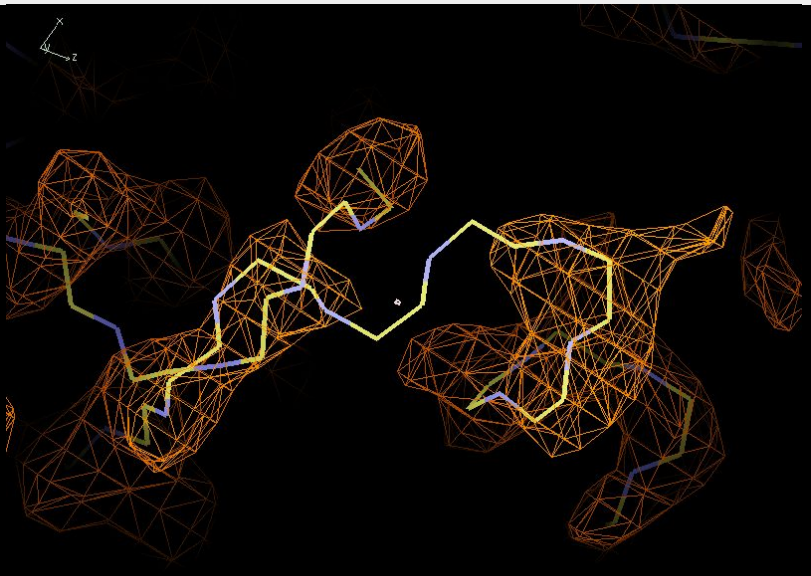
1. Find
2. Grow
3. Join
4. Link
5. ***Sequence***
6. Correct
7. Filter
8. NCS
9. Prune
10. Rebuild



We have skipped the linking step because there are no termini that can be linked in this view. The sequencing step works by assigning each residue a probability that it is each of the 20 residue types using the C-beta targets. These probabilities are then compared to the known sequence(s) to look for continuous runs that stand out with a high probability. In this example, the sequence only fits the chain well if an extra symbol is added in the middle. This is called an insertion and (unless the sequence is wrong) it means the chain has been built incorrectly. The reversed chain on the left wasn't sequenced as there was no part of the sequence that fits well.

Buccaneer - Example

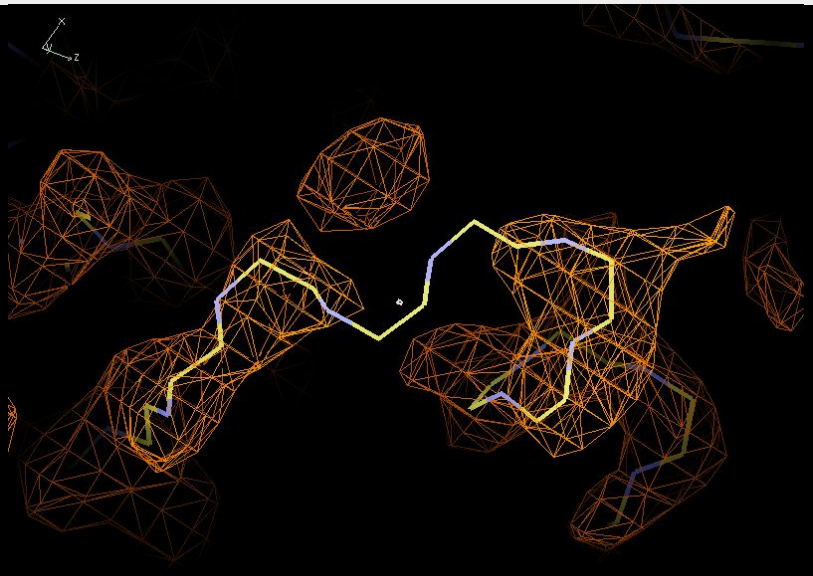
1. Find
2. Grow
3. Join
4. Link
5. Sequence
6. ***Correct***
7. Filter
8. NCS
9. Prune
10. Rebuild



The next step is to correct any insertions (where it has built one more residue) and deletions (where it has built one less residue) found during the sequencing step. The insertion that was over this loop as been corrected by removing three residues and rebuilding two.

Buccaneer - Example

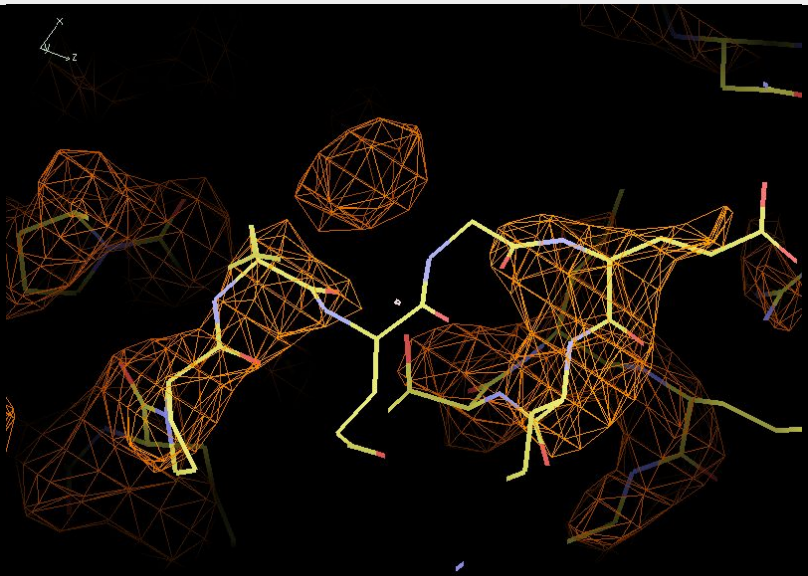
1. Find
2. Grow
3. Join
4. Link
5. Sequence
6. Correct
7. Filter
8. NCS
9. ***Prune***
10. Rebuild



The pruning step removes residues from clashing chains to produce a single consistent model. If the pruned chains have less than 6 residues remaining they are removed too. Pruning is done at this late stage to have the most information about which chain is correct. Residues that are unsequenced or are from shorter chains with worse fit to the density will be removed preferentially. In this case the reversed chain on the left was removed.

Buccaneer - Example

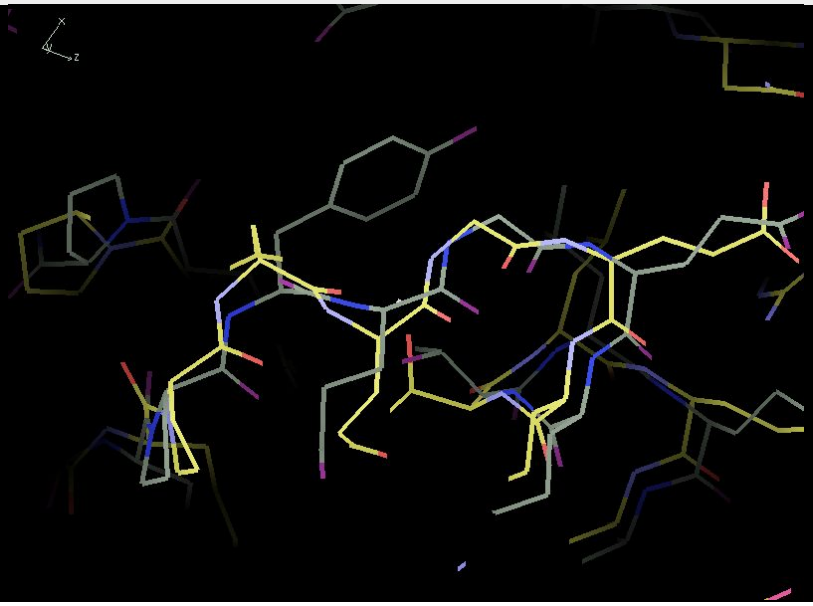
1. Find
2. Grow
3. Join
4. Link
5. Sequence
6. Correct
7. Filter
8. NCS
9. Prune
10. ***Rebuild***



Finally, the last step is to build side chains for each residue using a rotamer library. The rotamer with the highest average density at the atomic positions is chosen. If two side chains end up clashing, *Buccaneer* will try and find a pair of rotamers that does not clash. If this fails both residues will be truncated to C-beta.

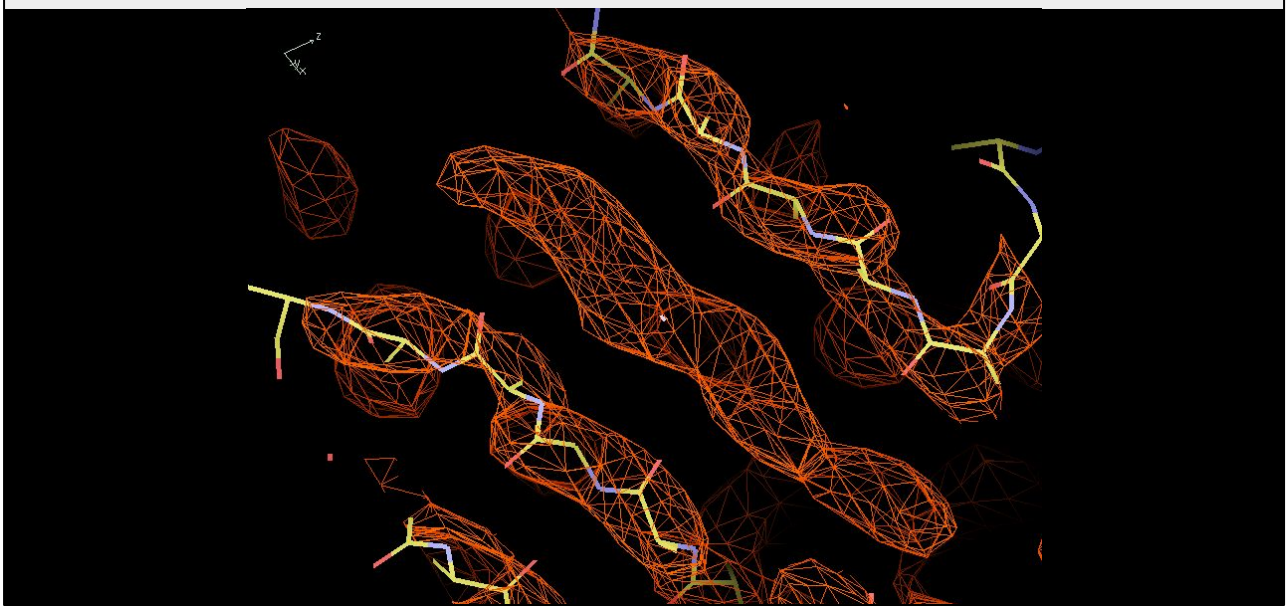
Buccaneer - Example

1. Find
2. Grow
3. Join
4. Link
5. Sequence
6. Correct
7. Filter
8. NCS
9. Prune
10. Rebuild



This shows a comparison with the deposited model. *Buccaneer* built most of the model quite well but got some things wrong, for example the the tyrosine side chain. The model will improve after refinement with *Refmac* and further cycles of *Buccaneer*.

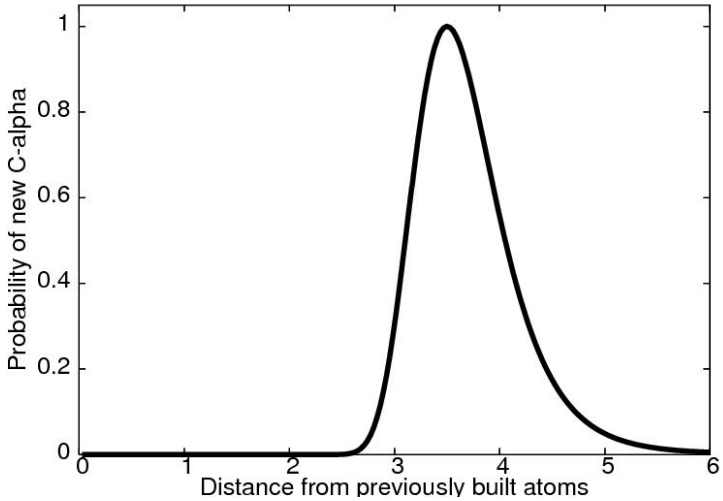
Buccaneer - Lateral growing



Buccaneer runs for multiple cycles where the 10 building steps are repeated. This example shows a model that was built in the first cycle, with un-modelled density for a beta strand that needs to be built. If the find step was repeated without giving it any new information about the residues that are already built it will find the same residue positions as before.

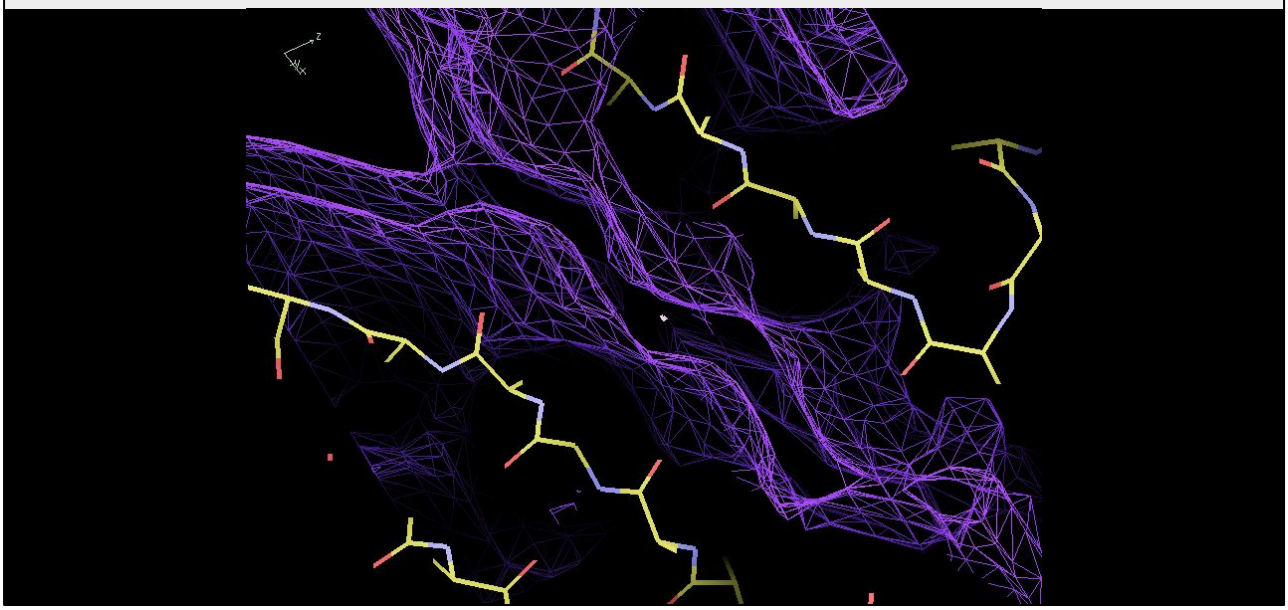
Buccaneer - Lateral growing

Build outwards
from existing chain fragments
by looking for
new Cα positions
at an appropriate distance
from the existing chain.



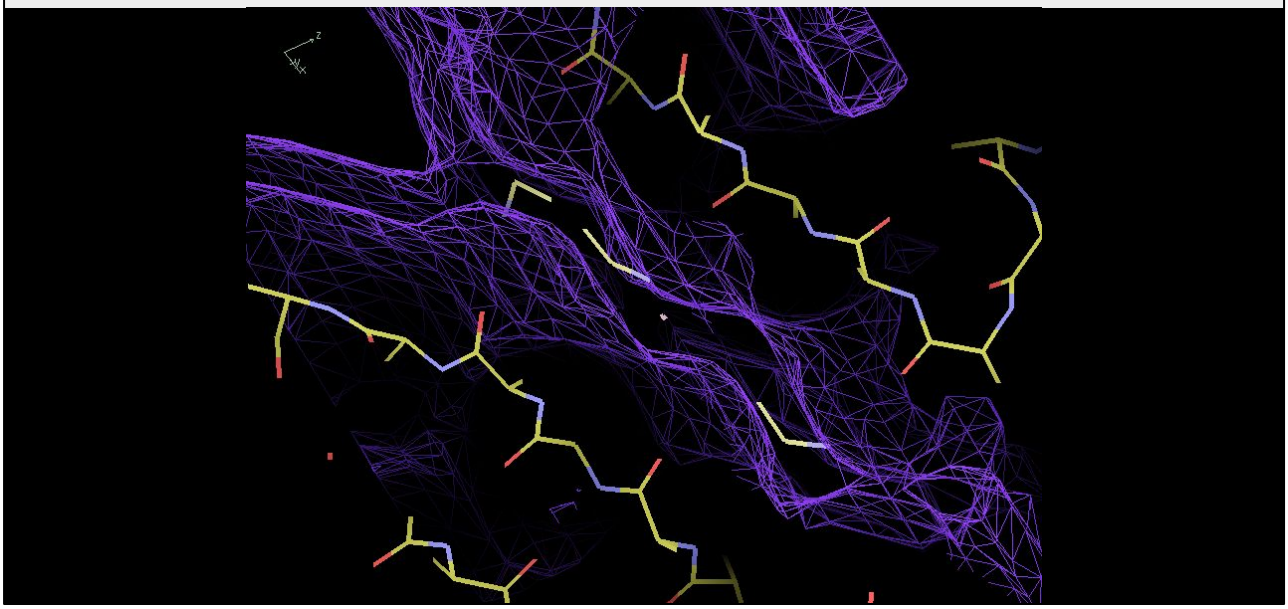
Buccaneer uses a concept called lateral growing that helps it find residues that are close (but not too close) to existing chains. Using data from published structures, it was found that the most likely distance of a C-alpha from another unbonded atom is around 3.5 A. The graph on the right shows the approximate probability distribution, which is combined with the model to produce a map with a prior probability of finding a C-alpha atom.

Buccaneer - Lateral growing



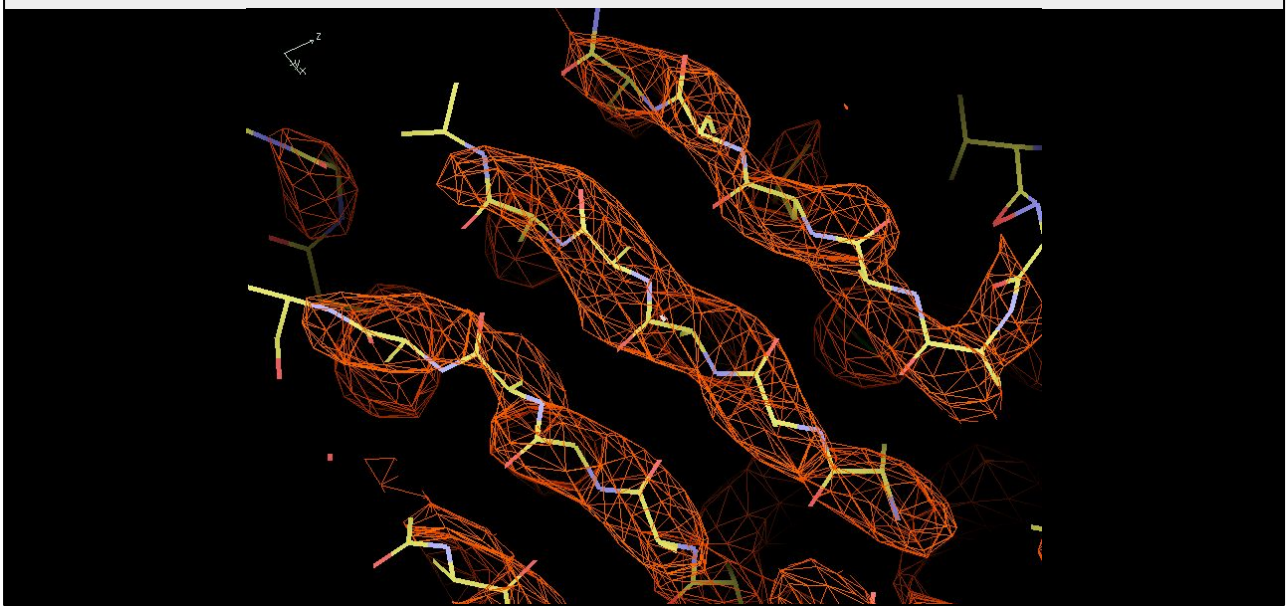
This is the prior probability map for the example we are looking at. The middle of the figure (where the unmodelled density was) has a high prior value because it is roughly 3.5 A away from both strands. Areas that are too close or too far away from the built model have a very low prior value so there is much less chance of finding new residue positions there.

Buccaneer - Lateral growing



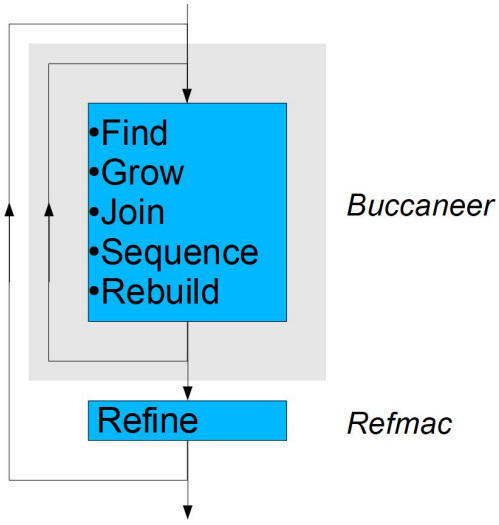
When the find step is repeated in the second cycle, potential residue positions are scored using the C-alpha target and the prior probability and three new residues were placed in the middle strand.

Buccaneer - Lateral growing



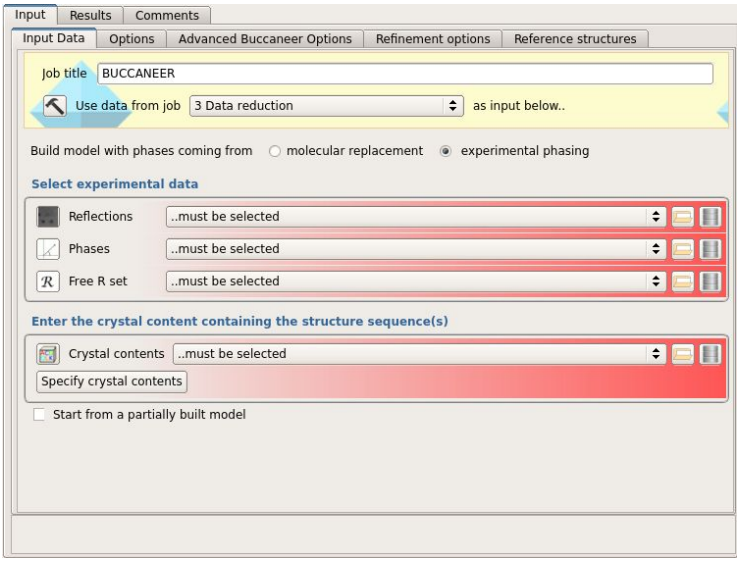
This is the model that comes out of the second cycle of *Buccaneer* after all 10 steps have been repeated.

Buccaneer - Pipeline



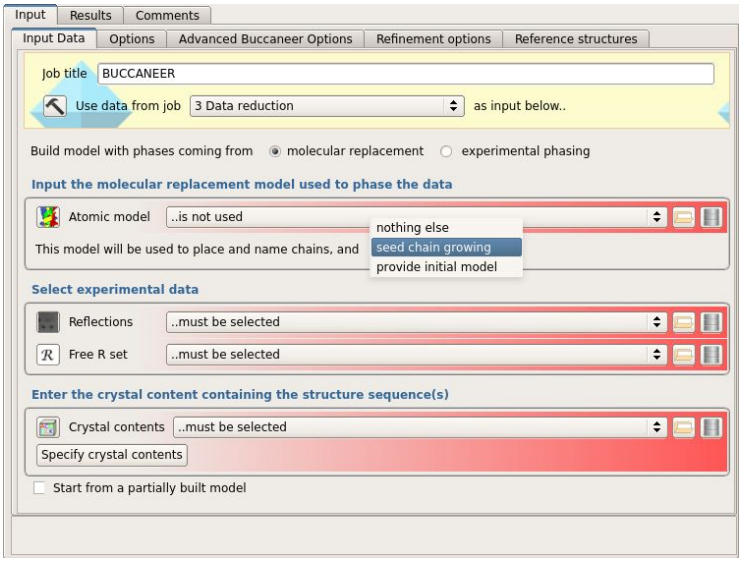
Buccaneer does not perform global refinement of the model, so when it is used through one of the interfaces (i.e. *CCP4i*, *CCP4i2* or *CCP4Cloud*) it actually runs a pipeline that combines *Buccaneer* with *Refmac*. This improves the model coordinates and B-factors and produces an updated (hopefully improved) map for further building. By default, the pipeline runs 2 or 3 cycles of *Buccaneer* followed by 10 cycles of *Refmac*, and repeats that 5 times. There have been some additional developments to the pipeline in *CCP4i2*, which now runs up to 25 cycles and also adds waters in *Coot* before refinement. There is also a new pipeline called *ModelCraft* that is covered later in this presentation.

Buccaneer - CCP4i2



This is the input page for *Buccaneer* in *CCP4i2* when the starting phases come from experimental phasing. You must provide reflections, initial phases, a free-R set and a description of the crystal contents. There is also the option to start from a partially-built model and more options in the other tabs.

Buccaneer - CCP4i2



This is the input page when the starting phases come from the refinement of a placed MR model. The MR model must be provided and you have the option to choose how the model is used, either just to place and name the chains that *Buccaneer* builds, or to provide initial residue positions as well.

Buccaneer - CCP4i2

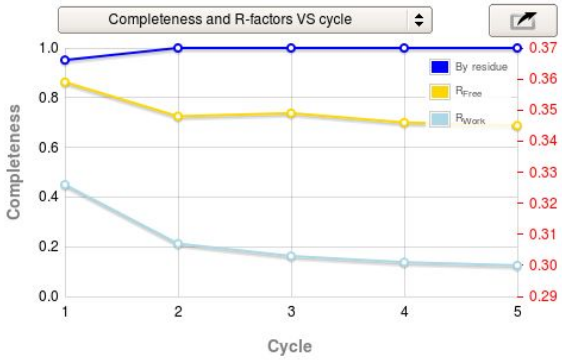
Results

118 residues were built in 2 fragments. Of these, 114 residues were assigned to the sequence.

The number of chains is estimated to be 1. Of these chains, 88.1% of the residues have been built.
Of the residues that were built, 100.0% were assigned to a chain.

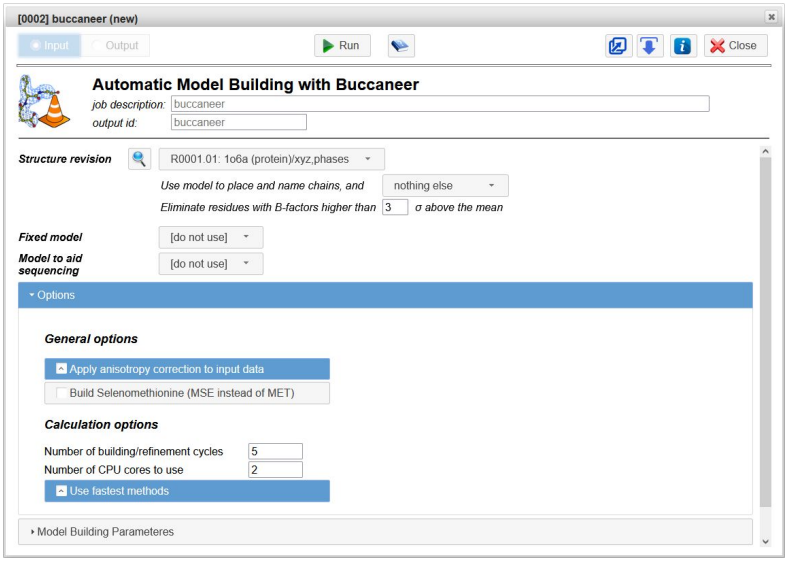
The refinement R-factor is 0.30, and the free-R factor is 0.34. The RMS bond deviation is 0.017 A.
On the basis of the refinement statistics, the model is approaching completion.

Completeness by residue	1.0
Completeness by chains	0.88
Number of chains	1
Residues built	118
Residues sequenced	114
Longest fragment	108
Number of fragments	2
R_{Work}	0.3
R_{Free}	0.345
RMS_{Bonds}	0.017
RMS_{Angles}	2.33



This is the results page in CCP4i2. It has a text description of the performance, a table of key figures and a chart showing R-factors and completeness per pipeline cycle.

Buccaneer - CCP4Cloud



This is the input page for the *Buccaneer* pipeline in *CCP4Cloud* with very similar options. *CCP4Cloud* also has a newer model-building pipeline called *CCP4Build*.

Buccaneer - Tips

Tidy up the structure in *Coot* afterwards:

- Connect up any broken chains
- Use density fit, rotamer analysis and other validation metrics
- Add waters, ligands, nucleic acids, sugars, etc
- Re-refine and look at the difference maps

If *Buccaneer* didn't do a very good job try:

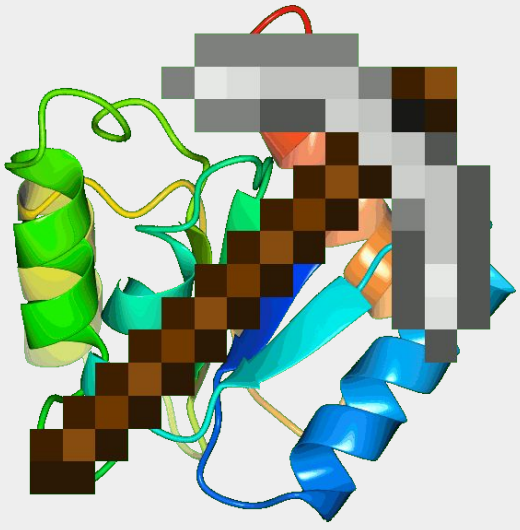
- Interactively pruning/building your model in *Coot* and re-submitting
- Improving the initial phases
- Adding non-incorporated heavy atoms (e.g. Fe) as known substructure
- Other model-building programs

Even if *Buccaneer* builds a fairly complete model there is still more work to do in Coot afterwards. This is an iterative process where you make some changes, re-refine the model then check it again. If *Buccaneer* hasn't done a very good job at building there are a number of things to try. The starting phases might be improved through density modification or by trying a different molecular replacement model. It is also worth trying different options, for example increasing the number of pipeline iterations. Other model-building programs could be used with the same input data or with the partially-built *Buccaneer* model. It is worth trying as many things as possible to increase the chance of finding something that works for your structure.

ModelCraft

Protein, RNA & DNA

paulsbond.co.uk/modelcraft



ModelCraft - Motivation

1. Improve the *Buccaneer* pipeline for molecular replacement structures
2. Build protein, RNA & DNA in a single pipeline
   (and other components in the future)
3. Share the new developments with all interfaces
   - Command line
   - CCP4i2
   - CCP4Cloud
   - CCP-EM

The original *Buccaneer* pipeline (with 5 iterations of *Buccaneer* and *Refmac*) showed good performance on experimental phasing structures. During my PhD I made a test set of over 1000 MR structures and we found a large number where the pipeline did not produce a very complete model. We wanted to make a better pipeline that could build more complete structures that included other programs, for example *Parrot* for density modification and *Nautilus* for building RNA and DNA. We also wanted to have the new developments shared between all interfaces without having to re-write them in multiple frameworks.

ModelCraft - Pipeline

Shift-field refinement (*Sheetbend*) - *Refmac*

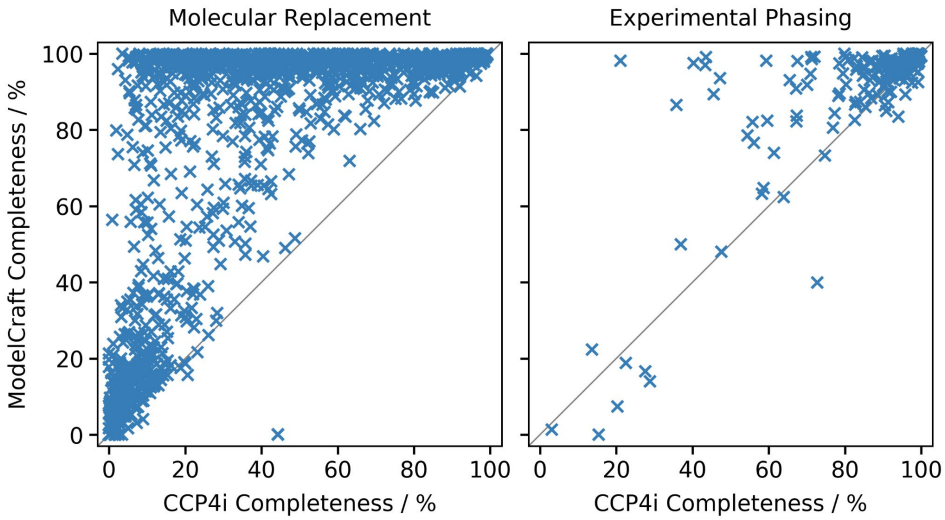
1. Prune chains and residues (*Coot*) - *Refmac*
2. Density modification (*Parrot*)
3. Add dummy atoms (*Coot*) - *Refmac*
4. Build protein (*Buccaneer*) - *Refmac*
5. Prune chains (*Coot*) - *Refmac*
6. Build nucleic acids (*Nautilus*) - *Refmac*
7. Add waters (*Coot*) - *Refmac*

Repeat up to 25 times by default

Rebuild side chains (*Coot*) - *Refmac*

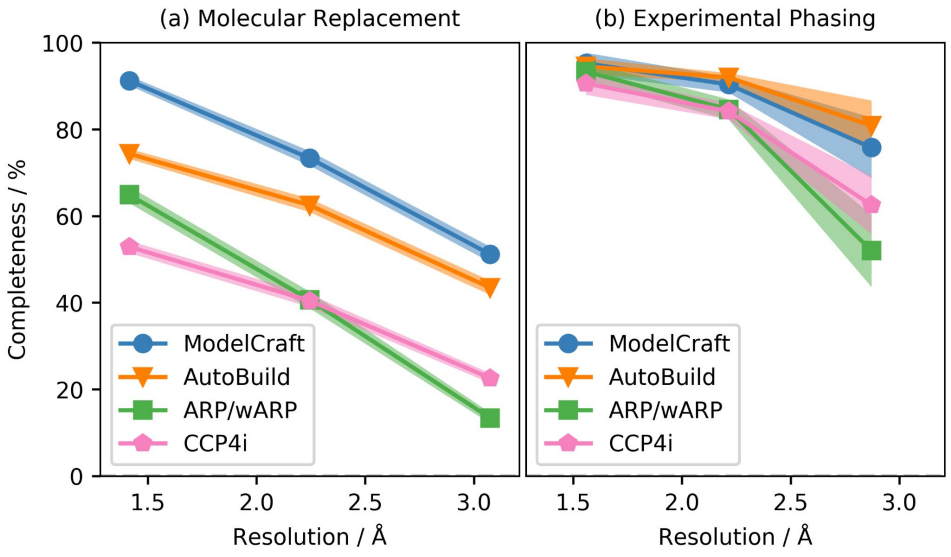
This is the overall *ModelCraft* pipeline as it stands. If you provide an starting model it is first refined using *Sheetbend* then *Refmac*. Then a single cycle of the pipeline has 7 steps, most of which are followed by refinement with *Refmac*. With high resolution data, it starts by pruning chains as well as individual residues and side chains. Then it uses *Parrot* for density modification, which is especially powerful if there is non-crystallographic symmetry that can be determined from the current model. After *Parrot*, it adds dummy atoms using *Coot* to improve the phases, but the dummy atom structure is only accepted if R-free improves. The dummy atoms are removed from the model then *Buccaneer* builds the protein and *Coot* is used to remove incorrect chain fragments. The pipeline also builds nucleic acids with *Nautilus* and finally adds waters with *Coot*, but again only accepting the result if R-free improves. Once R-free stops improving (or it reaches 25 cycles) the model from the cycle with the lowest R-free is used as the output. If the resolution is high then *Coot* is used to rebuild missing or incorrect side chains.

ModelCraft - Results



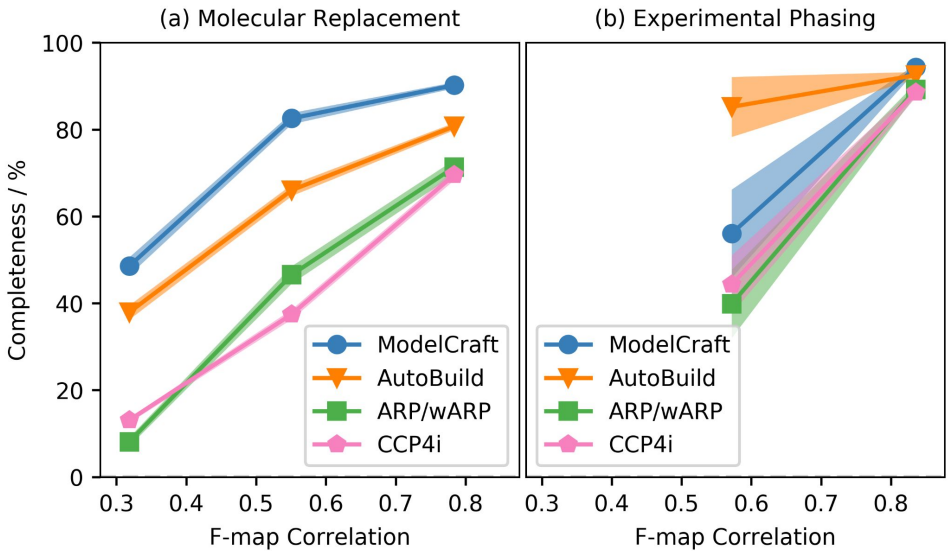
This is a comparison of the completeness of the protein model after the old *Buccaneer* pipeline from *CCP4i* and *ModelCraft*. Both were tested on 1348 MR datasets and 193 experimental phasing datasets. *ModelCraft* completes a lot more of the molecular replacement cases and nearly always does at least as well as *CCP4i*. Most of the experimental phasing cases are built well by both pipelines but *ModelCraft* still usually does better.

ModelCraft - Results



This is the average completeness when the datasets are split into three bins based on their resolution, along with the standard error as a shaded area. Results for PHENIX AutoBuild and ARP/wARP were provided by Emad Alharbi. The *CCP4i* pipeline and *ARP/wARP* perform similarly, but *ARP/wARP* is better at high resolution and *Buccaneer* is better at low resolution. *PHENIX AutoBuild* builds more complete models at all resolutions, but *ModelCraft* improves on this for molecular replacement cases.

ModelCraft - Results



F-map correlation is a measure of the quality of the phases. Again, *ModelCraft* builds more complete models from molecular replacement but *PHENIX AutoBuild* is better in difficult experimental phasing cases.

ModelCraft - Usage

Command line

```
$ pip3 install modelcraft

$ modelcraft --version
2.0.5

$ modelcraft xray \
> --contents contents.json \
> --data data.mtz \
> --model model.cif

$ modelcraft xray --help
```

Graphical User Interface (GUI)

COMING SOON

CCP4i2 and CCP4Cloud interfaces to be released after CCP4 8

ModelCraft can only be used on the command line currently. It can be installed using PIP for Python 3.7+. You have to set up the CCP4 environment so programs (such as cbuccaneer) can be used on the command line before running. There is more information on this at paulsbond.co.uk/modelcraft.

ModelCraft - Diamond Cluster

```
$ module load ccp4-workshop
$ clusterme
$ module load ccp4-workshop
$ module load python/3.9
$ pip install --user modelcraft
$ PATH=$PATH:$HOME/.local/bin
$ modelcraft --version
2.0.5
```

As well as running *ModelCraft* on your own machine, you can use these commands to run it on the Diamond cluster during the workshop. From a terminal on a login node, you need to load the workshop module, move to a cluster machine, then load the module again. You then need to load the Python module, install *ModelCraft* to your user space and set the path to the install location.

ModelCraft - Contents

Can be provided as sequences in FASTA or PIR format

OR

As a JSON format file with a description of the asymmetric unit contents

Easiest to start with (then edit) a contents JSON file for an existing PDB entry:

```
$ modelcraft-contents 1abc contents.json
```

If not known, the number of copies can be estimated:

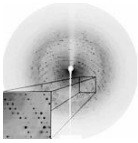
```
$ modelcraft-copies contents.json data.mtz
```

A description of the asymmetric unit contents needs to be provided to the --contents argument. This can be a sequence file, but this has the downside that the number of copies may be guessed incorrectly leading to the wrong solvent content in *Parrot*. The JSON format allows for a more accurate solvent calculation by specifying the number of copies as well as including other components such as sugars and ligands.

ModelCraft - Data

Reflection data in MTZ format containing

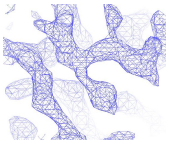
Observations



F-mean, F-anom,
I-mean or I-anom

Free-R flag

Phases



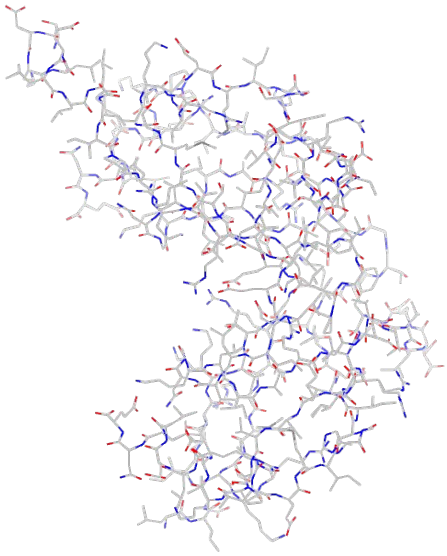
Phi-FOM or ABCD

Only if starting without a model, or if starting phases shouldn't come from model refinement

An MTZ must be given to the --data argument that must contain observations and a free-R flag. The column names for these don't need to be provided unless there are multiple possible columns to choose from. If you provide a model but don't want to start from the model phases then column names must be passed to the --phases argument.

ModelCraft - Model

- Can be provided in mmCIF, mmJSON or PDB format
- Needs to be placed correctly in the ASU as MR will not be performed
- Doesn't need to be refined as *Sheetbend* and *Refmac* are used
- Residues other than protein, RNA, DNA or water will be kept fixed



A starting model can be provided using the --model argument. It may be useful to use jelly-body refinement first if the combination of *Sheetbend* and *Refmac* are not doing a good job at refining the input model. If you know the location of non-incorporated heavy atoms, it can be useful to include these to improve the phases and to stop the model being traced through that density.

Nautilus

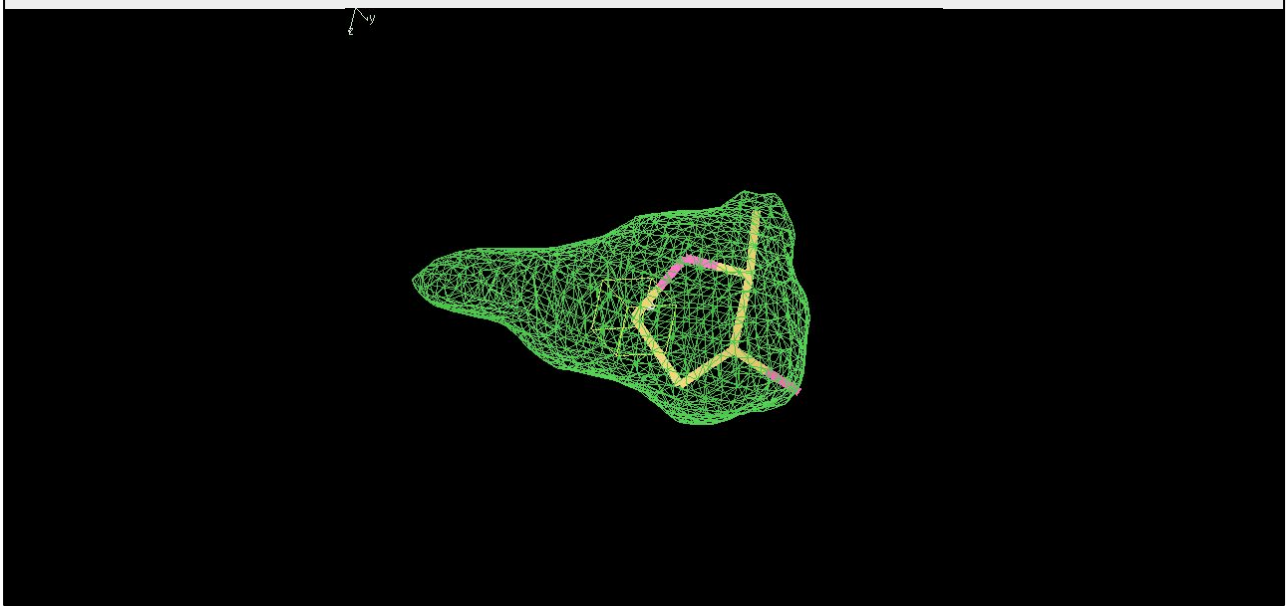
RNA & DNA

Nautilus - Fingerprints

- Identify high and low density features consistent with the presence of nucleic acid features
- Very fast
- Related to *Essens* (Kleywegt and Jones), but looks at both ridges and troughs
- Fingerprints for sugar, phosphate and base type (A/C/G/U)

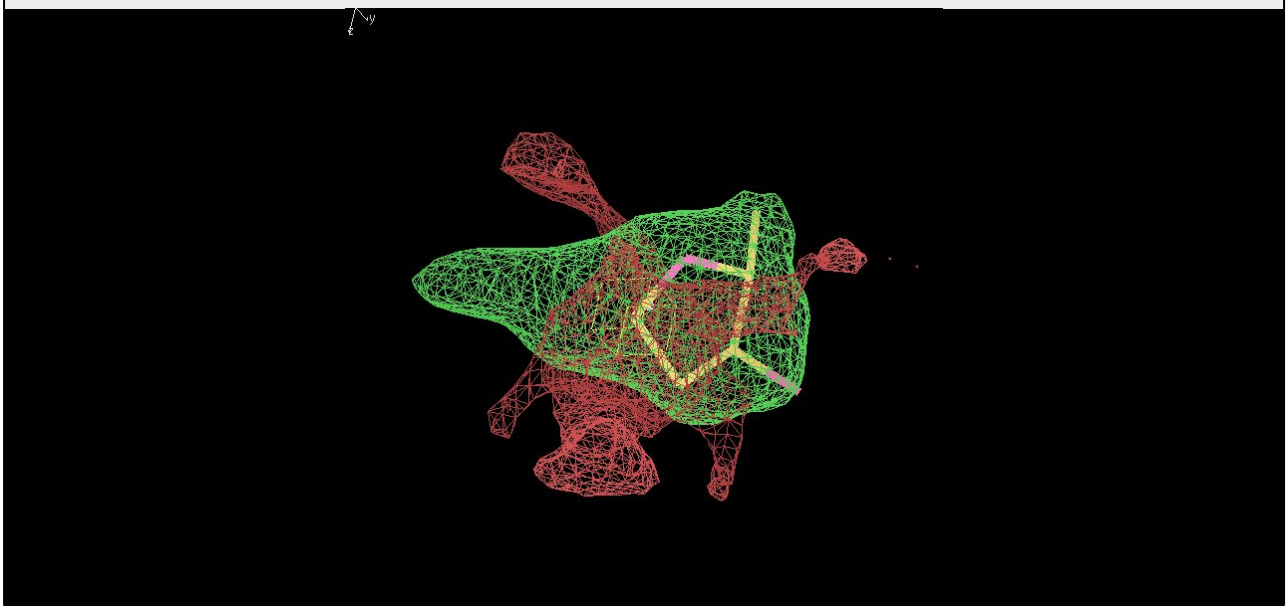
Like *Buccaneer*, *Nautilus* also identifies residue positions by looking for patterns of conserved high and low density, but it does so using a fingerprint detection method instead of simulating a map for a reference structure. The sugar and phosphate fingerprints are used for building the backbone, and the base fingerprint is used for sequencing.

Nautilus - Fingerprints



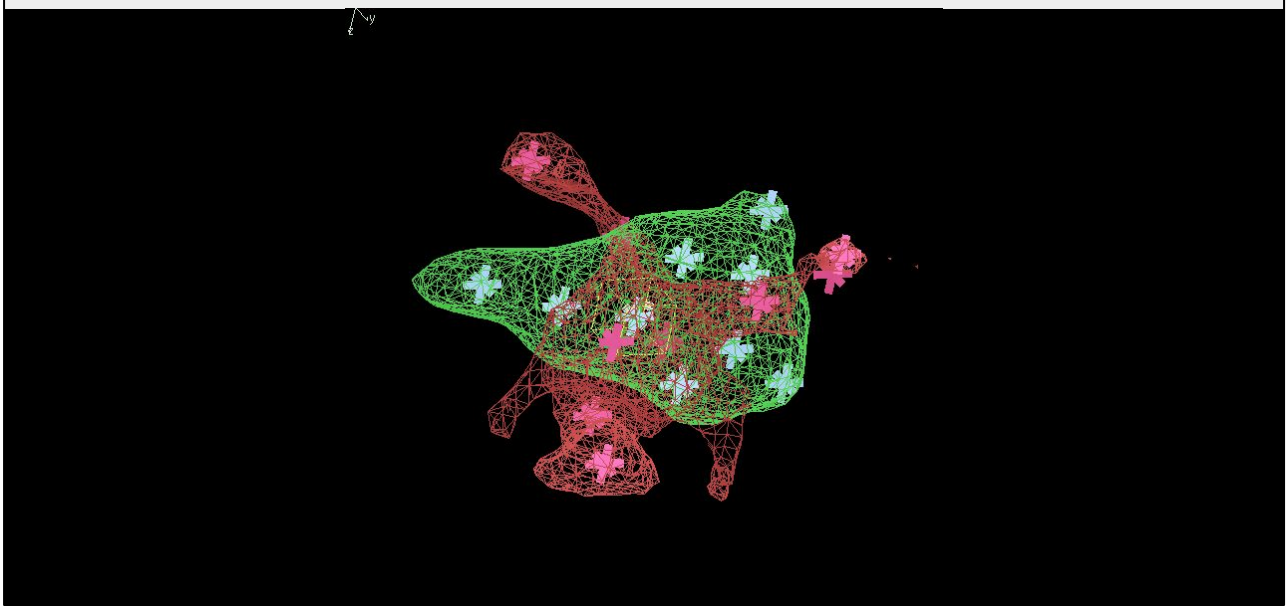
This shows how the sugar fingerprint was formed. Lots of sugars in a were superposed over this one in a standard orientation. The green map is the minimum density for all of the overlapped sugars. The minimum density around the sugar is high because it's always there. The first part of the density for the base is also conserved. The phosphate positions aren't visible because they have multiple possible conformations.

Nautilus - Fingerprints



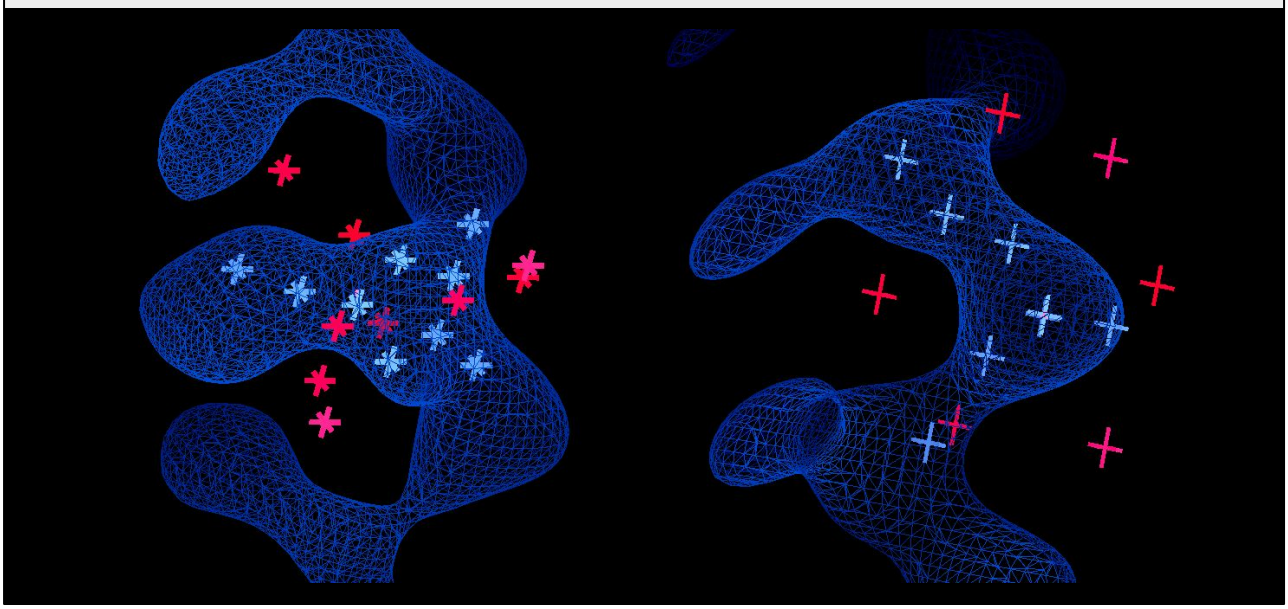
The red map is the negative of the maximum density for the overlapped sugars, so these are the regions where we never see much density.

Nautilus - Fingerprints



The fingerprint is then made up of high probe points where we expect high density and low probe points where we expect low density. High probe points were put at the atom positions plus one more at the start of the base. Low probe points were put in the largest red peaks until there were the same number of high and low probes. This ends up being a fast method because it's only looking at a few key points.

Nautilus - Fingerprints



The same process was followed to create a fingerprint for phosphate. It has high probe points on the phosphate atoms and in the neighbouring sugars and low probe points in the surrounding regions. This shows the sugar and phosphate fingerprints in place over some example density.

Nautilus - Fingerprints

Fast Score

Minimum of the high probe points minus the maximum of the low probe points.

Used initially to search all positions and orientations.

Can only get worse as you include more probe points so the calculation can be stopped early.

Accurate Score

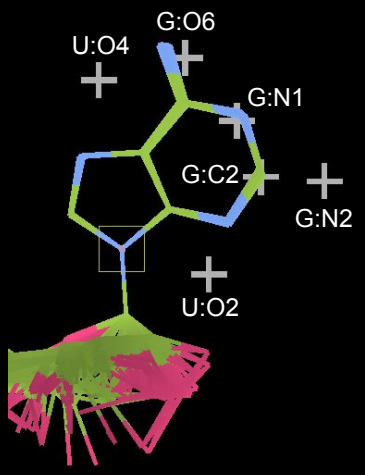
Mean of the high probe points minus the mean of the low probe points.

Used on the initial list of positions found using the fast score.

There are two methods used for scoring a position in the map using the sugar and phosphate fingerprints. The first is the fast score that uses the minimum high probe minus the maximum low probe. It is faster because it can be stopped if the score isn't good enough before density for all the points is calculated. The second accurate score is only calculated for the positions with the best fast score.

Nautilus - Fingerprints

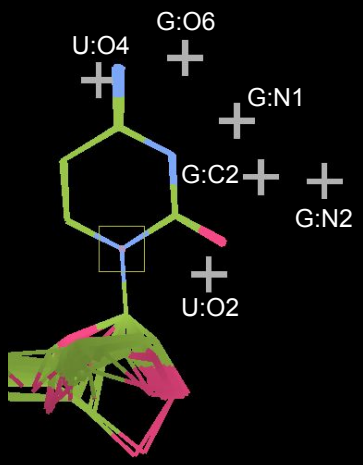
	U:O4	U:O2	G:O6	G:N1	G:C2	G:N2
Adenine	-	-	+	+	+	-
Cytosine						
Guanine						
Uracil						



The base fingerprint has 6 probe points that were chosen to try and distinguish between different base types. Depending on the base type, you will expect either high or low density at a particular point. For example, adenine would have high density at G:O6, G:N1 and G:C2 and low density at U:O4, U:O2 and G:N2. This is represented in this table by pluses and minuses.

Nautilus - Fingerprints

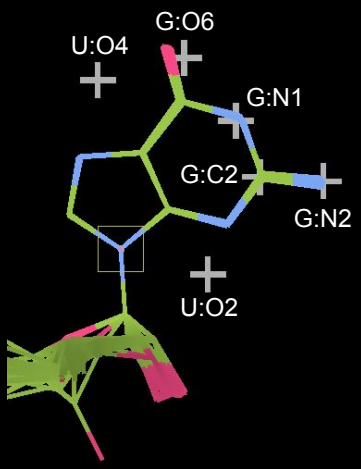
	U:O4	U:O2	G:O6	G:N1	G:C2	G:N2
Adenine	-	-	+	+	+	-
Cytosine	+	+	-	-	-	-
Guanine						
Uracil						



Cytosine only has high density at U:O4 and U:O2.

Nautilus - Fingerprints

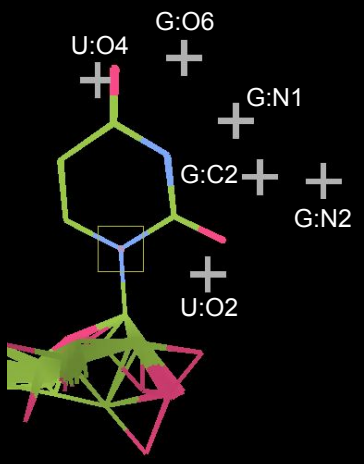
	U:O4	U:O2	G:O6	G:N1	G:C2	G:N2
Adenine	-	-	+	+	+	-
Cytosine	+	+	-	-	-	-
Guanine	-	-	+	+	+	+
Uracil						



Guanine is similar to adenine but it also has density at G:N2.

Nautilus - Fingerprints

	U:O4	U:O2	G:O6	G:N1	G:C2	G:N2
Adenine	-	-	+	+	+	-
Cytosine	+	+	-	-	-	-
Guanine	-	-	+	+	+	+
Uracil	+	+	-	-	-	-



Uracil covers the same points as cytosine so it is hard to tell between these two bases. However, it is much easier to tell the difference between the small pyrimidine and the large purine bases.

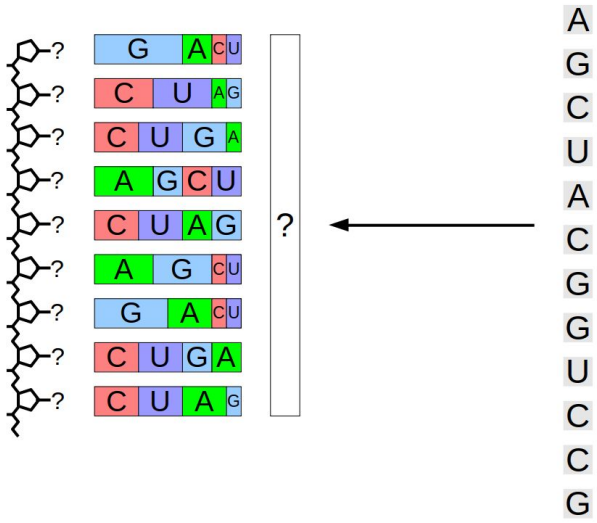
Nautilus - Fingerprints

Get the probabilities of each base type using **k-Nearest Neighbour**

1. Calculate z-scored densities for the 6 sample positions for 200 bases (50 of each type) to form a sample database.
2. Calculate z-scored densities for the 6 sample positions for the unknown base.
3. Find the 50 closest matches to the unknown base from the database.
4. Assign the probability of A/C/G/U based on the proportion of the 50 closest matches that are each type (with an error term).

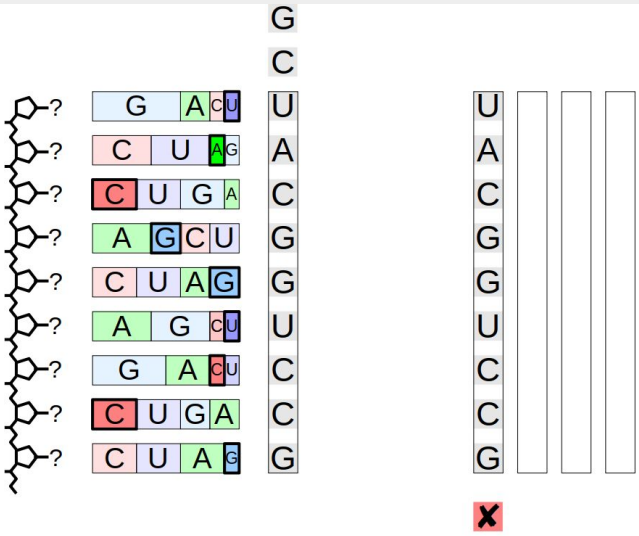
For this fingerprint, probes aren't simply classified as having high or low density. The aim is to get a probability of each base type, which is done using the k-Nearest Neighbour (k-NN) algorithm. The final probability is based off the number of matches of each type along with how close each match was.

Nautilus - Fingerprints



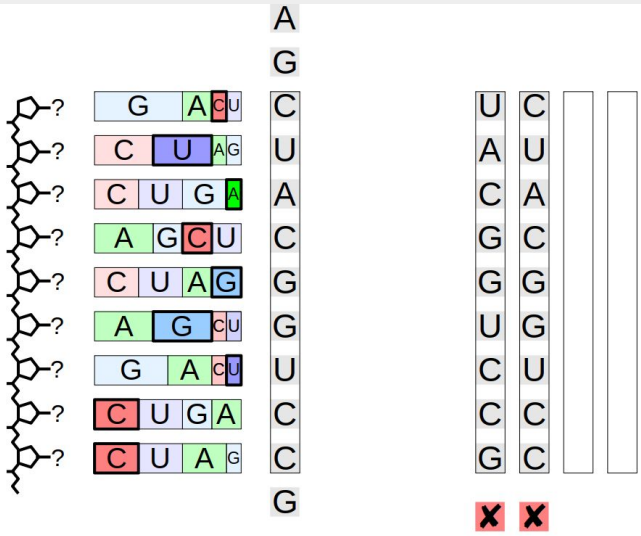
Once we have 4 probabilities for each base in the chain, we need to find out which part of the known sequence it corresponds to. In this diagram the probability is represented by the width of the box. C and U have the same probability because they have very similar density.

Nautilus - Fingerprints



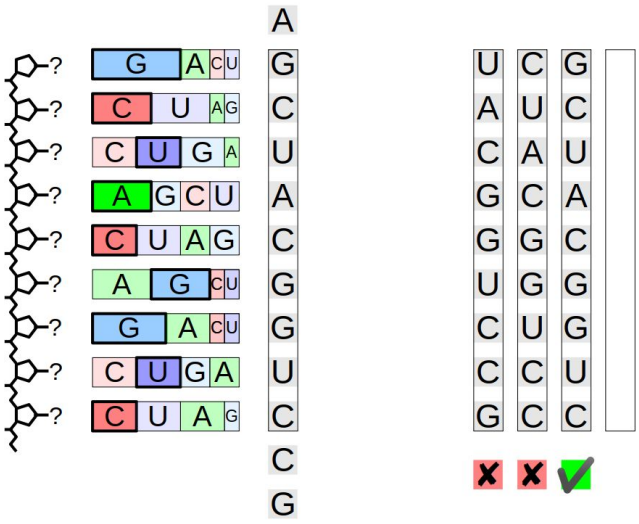
Putting the sequence against the chain gives some bases with high probability and some with low probability.

Nautilus - Fingerprints

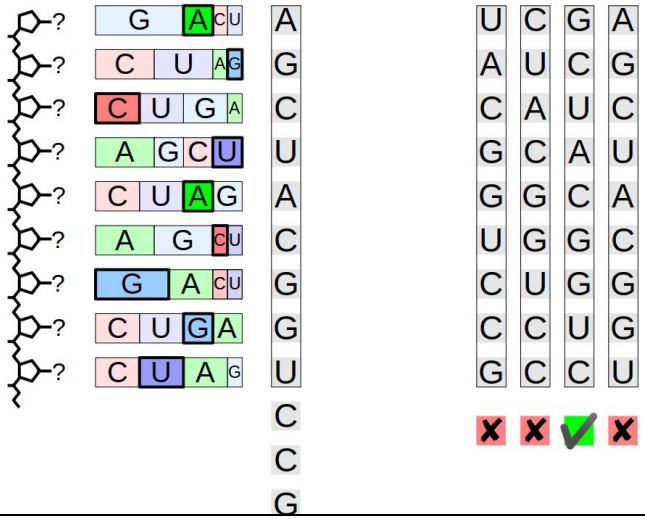


The sequence gets moved along the chain to score each possible assignment.

Nautilus - Fingerprints



Nautilus - Fingerprints



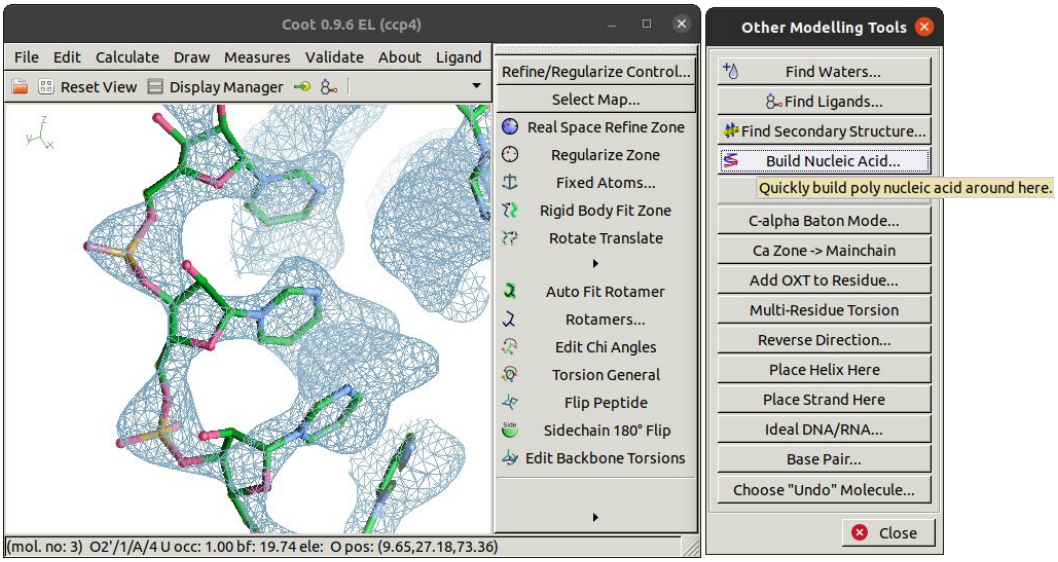
After looking at all the possibilities the third sequence stands out as having a higher probability so that one is chosen. Sometimes none of the assignments will stand out and the chain will remain unsequenced. This can happen if it is built incorrectly or the density is very poor.

Nautilus - Steps

1. Find sugar and phosphate positions
2. Grow them into chain fragments
3. Join overlapping fragments and resolve branches
4. Link nearby termini
5. Prune residues to resolve clashes
6. Rebuild chains to ensure connectivity
7. Sequence the chains
8. Build bases

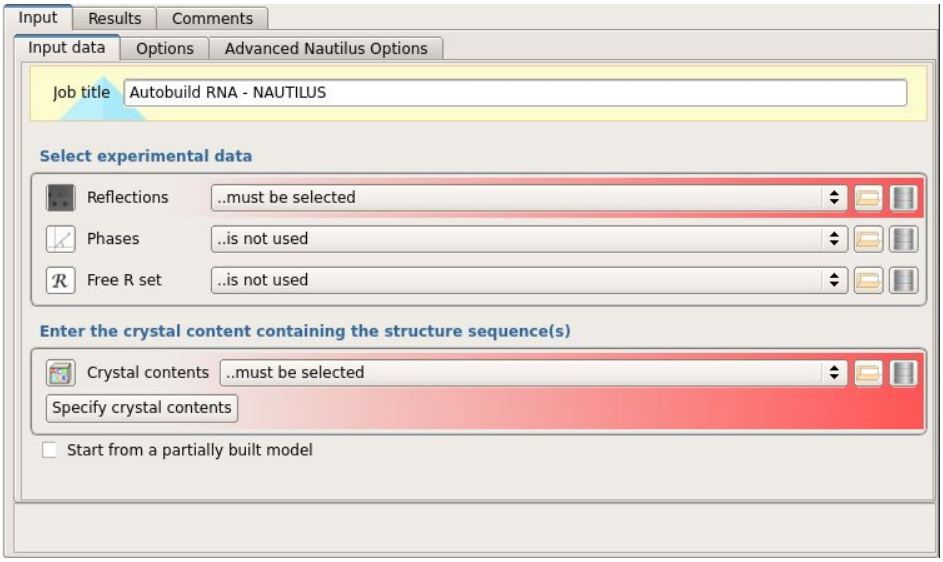
The overall structure of the program is similar to that of *Buccaneer*, with steps to find initial positions, grow them into chain fragments, then join and link the chains. There are some differences however: the pruning step is done before assigning the sequence and a rebuilding step was added to ensure the whole chain continues smoothly without unrealistic geometry.

Nautilus - Coot



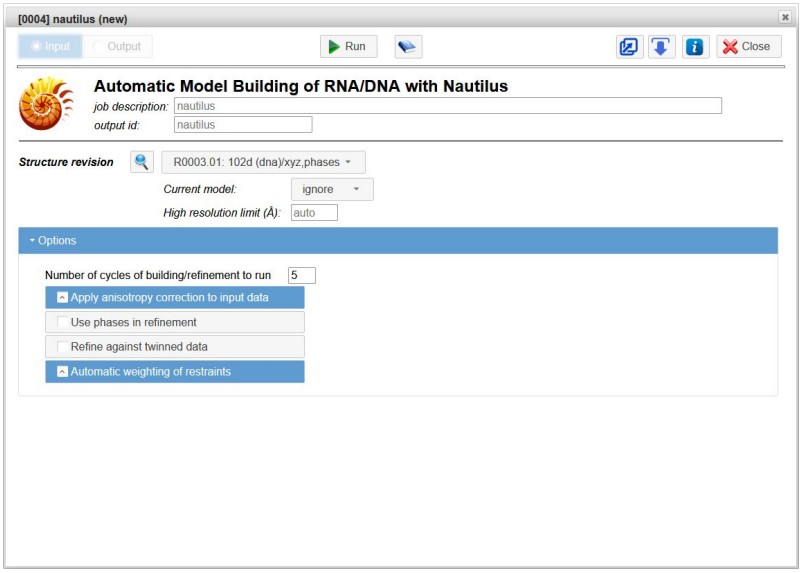
One way of using *Nautilus* is interactively through *Coot*. This version (*Cootilus*) does not include the sequencing step so it will just build poly-uracil near the cursor. It tries to improve existing nucleic acids as well as building new residues. The built model then needs to be refined and mutated to the correct sequence. *Coot* also contains *RCrane* tools for more interactive nucleic acid building.

Nautilus - CCP4i2



This is the input page in CCP4i2, which runs a pipeline that iterates *Nautilus* with *Refmac* for refinement similar to the *Buccaneer* pipeline.

Nautilus - CCP4Cloud



This is the interface to the *Nautilus* pipeline in *CCP4Cloud*.

Acknowledgements

Kevin Cowtan

Keith Wilson

Emad Alharbi