Product-Mix Auction User's Guide DRAFT

Release 0.1

Well-Typed LLP

Sep 30, 2019

CONTENTS

CHAPTER

ONE

INTRODUCTION

The Product-Mix Auction is a single-round sealed-bid auction for multiple units of multiple distinct goods. This package contains the following:

- an implementation of the core Product-Mix Auction algorithm and various extensions as a Haskell library;
- a command-line program **`pma`**, which reads a complete specification of an auction from the command line and input files, runs the auction and outputs the results (see *Running auctions via the command-line interface*);
- a web application **`pma-server`**, which provides a single-user interface allowing an auction specification to be constructed in a web browser form, sent to the server, solved and the results presented back in the web browser (see *Running auctions via the web app*).

The remainder of this section gives a high-level overview of the Product-Mix Auction algorithm and the key concepts involved. Subsequent sections explain how to make use of the programs to run auctions. For further background information, see The Product-Mix Auction: a New Auction Design for Differentiated Goods. For a more precise description of the linear programmes being solved, see the accompanying document *"Specification for Implementing the Product-Mix Auction Solver"* on http://pma.nuff.ox.ac.uk.

The core Product-Mix Auction algorithm takes as input a supply specification and a set of bids (possibly from multiple bidders). It converts these into a linear programme, and solves it to produce a price for each distinct good and an allocation of goods to bids. (Multiple runs of linear programmes may be necessary for some features.) The number of (distinct) goods is fixed throughout.

1.1 Supply

The *supply specification* describes the quantities available of each (distinct) good, and the sequence of reserve prices that the auction price must exceed for the corresponding quantity to be available. Typically the supply specification will be chosen by the auctioneer in advance of the auction. It consists of a supply curve for each good, and a supply ordering across the goods.

A *supply curve* is a generalisation of the concept of a reserve price for a good: it gives multiple reserve prices depending on the number of units allocated. The price of the good will be at least the reserve price given by the supply curve for the quantity of the good that has been allocated. Supply curves are given as step functions, specifying the width of each step (number of units available) and the price at which they are available. The simplest case is a supply curve with a single step, which corresponds to there being a fixed number of units available for the good at a single reserve price.

A *supply ordering* gives the relationship between different goods:

- *Vertical* supply ordering means that each good is more valuable than its predecessor. Prices in the supply curve are given relative to the predecessor's price (or to zero, for the first good). Quantities in the supply curve represent the total amount available for the good and any successive goods. Thus the supply curve for the first good gives the total quantity available of all goods. For example, with two goods, the supply curve for good 1 has absolute prices for good 1 based on the total quantity of both goods, while the supply curve for good 2 has prices relative to good 1 for the quantity of good 2.

- *Horizontal* supply ordering means that goods are independent. All prices are given relative to zero. Quantities in the supply curve represent the amount available for that good alone. For example, with two goods, the supply curve for good *j* has absolute prices for the quantity of good *j*.
- More general *tabular* orderings are possible, with goods organised into multiple columns (having related prices and quantities as with a vertical ordering), but horizontal relationships between the columns. A tabular ordering may also have a special good below all the columns, whose supply curve gives the total quantity available of all goods.

A *base good* for a supply ordering is one whose supply curve prices are given relative to zero, and whose quantities include the amounts allocated to successive goods. Thus a vertical ordering has the first good as its sole base good, while for a horizontal ordering, all goods are base goods.

The *auction size* is the total width of all supply curves for base goods. Thus a vertical ordering has auction size equal to the total width of the first good's supply curve, while a horizontal ordering has auction size equal to the total of the widths for all the supply curves.

See *Input format for supply* and *Supply specification in the web app*.

1.2 Bids

A *bidder* is represented to the program as a list of their bids. The bidder who made a bid is not relevant to the basic functioning of the algorithm, but makes a difference to the presentation of the output and to *Additional constraints*.

In the simplest case, a *bid* consists of a single quantity, and a non-negative per-unit price offered for each good. A bid is said to be *paired* if it offers a non-zero price for two or more goods, or *single* if it offers a non-zero price for only one good.

If the auction prices are strictly above the bid prices for all goods, the bid is unsuccessful. Otherwise, the bid receives an allocation of its preferred good(s) (i.e. the good(s) for which the bid price minus the auction price is maximised), up to the quantity requested. If there are multiple preferred goods, the bid may receive any combination of them. If the bid price exactly equals the auction price for the preferred good(s), the bid is rationed and may receive fewer units than it requested.

For example, suppose the auction determines prices of 5 for good A and 6 for good B. A bid for 10 units of (good A at price 5) or (good B at price 8) will receive 10 units of B. A bid for 1 unit of good A at price 5 (only), may receive no units, a whole unit, or some intermediate fraction. A bid for 1 unit of (good A at price 3) or (good B at price 5) will receive no units.

An *asymmetric bid* allows bidders to express more general trade-offs than 1-1 between goods. For example, an asymmetric bid might demand (12/2 units of good A at price 5) or (12/3 units of good B at price 8). If this bid wins, depending on the auction prices, it might receive 6 units of good A, 4 units of good B, or some linear combination in those proportions (i.e. any x units of A and y units of B such that $2x + 3y = 12$).

A *generalised bid* includes a maximum quantity for each good, as well as the overall quantity. For example, a generalised bid might demand 10 units of (up to 7 units of good A at price 80) or (up to 8 units of good B at price 50). If the auction prices were zero, this bid would receive all 7 units of good A (because it prefers A to B as expressed by the higher bid price) and the remaining 3 units of B (to exhaust the overall quantity).

If enabled in the auction configuration, bids may optionally be both asymmetric and generalised.

See *Input format for bids* and *Bids in the web app*.

1.3 Total Quantity Supply Schedule

A Total Quantity Supply Schedule (TQSS) is a function from prices to the total number of units (of all goods) that should be made available at those prices. This allows the auctioneer flexibility in deciding how many units to make available depending on demand. Typically the TQSS function is specified as a list of steps, where the width represents the quantity and the height represents the price. "Solving" the TQSS means finding the intersection between the supply function specified by the user and the demand function calculated from the bids.

Prices used in the TQSS may be normalised (i.e. expressed relative to the applicable supply curve step) or absolute. This affects the algorithm used for solving the TQSS, and the options that are available for use with it, as described in the following subsections. See also *Input format for TQSS* and *TQSS in the web app*.

1.3.1 TQSS with absolute prices

With absolute prices, a measure on prices must be specified by the auctioneer, converting the prices of all the goods into a single price for use in the TQSS. Typically this will be the mean price of all the goods, or the price of a single nominated good. To solve the TQSS, the software will solve the basic linear programme repeatedly, adjusting the total quantity available each time and calculating the resulting price measure, to produce a demand curve. The (approximate) point where the TQSS function intersects the demand curve will give the total quantity and price measure, and the auction prices and allocations will be presented for this total quantity.

There are two possible approaches to limiting the total quantity available:

- **Option "Scale supply"**: Scaling the supply curves so that the auction size is changed to the desired limit. (This *increases* the quantities sold to more than the quantities offered by the originally-specified supply curves, if demand is sufficiently high.)

  A parameter λ between 0 and 1 must be specified. Supply curve quantities for base goods are scaled in proportion to the new total auction size. Quantities for non-base goods (i.e. goods 2 and up for a vertical auction) are scaled similarly if $\lambda = 0$, not scaled at all if $\lambda = 1$, or partially scaled for intermediate values of λ. The cumulative totals of the supply curve step widths are rounded upwards; this may introduce rounding errors. The original auction size will be the minimum bound of the search (unless explicitly overridden by the user), so supply curves will typically be scaled upwards, not downwards. The given TQSS function must have a first step of height zero and width equal to the auction size.

- **Option "Constrain supply"**: Adding a constraint to the linear programme requiring that the total quantity actually supplied to all goods does not exceed the limit. (This *decreases* the quantities sold to less than the quantities offered by the originally-specified supply curves, if demand is sufficiently low.)

  The original supply curves are not changed, so the maximum possible quantity available will be limited by the original auction size. The parameter λ is not relevant.

When scaling the supply curves with a non-zero value for the parameter λ, the supply ordering must be vertical. Otherwise, any supply ordering may be used for a TQSS with absolute prices.

1.3.2 TQSS with normalised prices

With normalised prices, the TQSS is solved using a single run of a modified linear programme (encoding the structure of the TQSS in the constraints). The supply ordering must be horizontal, and the TQSS constrains the total supply available (the original supply curves are not scaled).

1.4 Additional constraints

The auctioneer may specify additional constraints on the quantities received by bids. These are directly implemented as constraints in the linear programme.

The form of additional constraint that is currently supported is a maximum quantity that may be allocated (across all goods) to any one bidder. This may be an absolute maximum, or expressed as a fraction of the auction size.

See *Additional constraint options* for how to specify additional constraints in the command-line interface. It is not currently possible to specify additional constraints in the web interface.

1.5 Rationing and rounding

Rationing is the process of adjusting the results of the basic Product-Mix Auction to "fairly" share between bidders. Multiple rationing schemes are supported by the software.

The standard approach to rationing is to run the basic auction twice. The first run makes it possible to identify multiply-marginal paired bids (those that are indifferent between receiving multiple goods). On the second run, such bids are "smeared out" by replacing the bid with a collection of bids approximating a linear demand decreasing from the bid quantity to zero as the price increases. This means that the auction will allocate rationed goods (approximately) fairly to multiply-marginal bids. Subsequently, goods are fairly shared between singly-marginal bids (those that are indifferent between receiving or not receiving a single good). This approach may slightly favour paired bids, in the sense that a paired bid at the auction prices may receive an allocation in preference to a single bid at the auction prices.

Optionally, this "smearing out" process can be applied to all marginal bids, not only those that are multiply-marginal. This does not favour paired bids, but it may require more computational time.

Alternatively, rationing can be disabled, in which case rationed goods will be allocated arbitrarily to marginal bids, and identical bids may not receive identical allocations. Of course, the constraints of the auction will be satisfied, so non-marginal bids will always receive the correct allocation. In addition, we have the facility to randomly reorder bids before constructing the linear programme, thereby ensuring a kind of equity even without rationing (as favoured bids will be determined by chance rather than by the structure of the LP or implementation details of the LP solver).

Regardless of the rationing scheme in use, allocated quantities are calculated and reported to a limited number of decimal places, controlled by the user. Increasing the number of decimal places yields more precise results but may require more computational time. Since floating-point arithmetic is used internally, very large numbers of decimal places may lead to inaccurate results. In any case, as a result of rounding there may be a small over-or-under-allocation in the total quantity allocated by the auction (e.g. if the exact solution results in allocations of 0.25 and 0.75 with a total of 1 unit, rounding to 1 decimal place yields 0.3 and 0.8 with a total of 1.1 units).

See *Rationing and rounding options* for how to control rationing in the command-line interface, and *Rationing in the web app* for the controls provided by the web app.

1.6 Maximisation strategies

The basic Product-Mix Auction supports a choice of objectives: maximising efficiency (the combination of the bidders' surplus and the seller's profit) or maximising profitability (for the seller alone).

Maximising efficiency determines auction prices and allocations that give the most efficient outcome for the auctioneer and all bidders, following the standard approach described in *"Specification for Implementing the Product-Mix Auction Solver"* on http://pma.nuff.ox.ac.uk. All the options described above are supported.

Maximising profitability determines auction prices and allocations that respect the bidder's constraints and maximise the auctioneer's profit. This uses an experimental brute-force algorithm based on that used for budget-constrained bids (see *Budget constraints*). In this case, the supply ordering must be horizontal, bids must be simple (not generalised or asymmetric), a TQSS or additional constraints may not be used, and rationing will not be performed. The prices and allocations that maximise profitability may not be unique, and in this case an arbitrary solution will be reported.

See also *Maximisation strategy options* and *Maximisation strategies in the web app*.

CHAPTER

TWO

INSTALLATION

The following sections explain how to install the software from source, for each supported platform. An installation program for Windows and an app bundle for Mac OS X are also available, and can be used to install pre-compiled binary versions of the software without the need to follow the instructions below.

2.1 Installation on Linux

The package depends on the GNU Linear Programming Kit (GLPK) (versions 4.45 and 4.57 are known to work), BLAS and LAPACK. On Debian/Ubuntu-based systems, the required packages are `libglpk-dev`, `libblas-dev` and `liblapack-dev`. On Fedora/CentOS/RHEL-based systems, the required packages are `gplk-devel`, `blas-devel` and `lapack-devel`.

The **`git`** VCS client is required to download the source code. To install from source, a Haskell toolchain including either **`ghc`** 8.0.2 and **`cabal-install`** 2.0, or including **`stack`** 1.6.1, is required. (Later versions should also work.) The following subsections provide instructions for installing the package using either **`cabal-install`** or **`stack`**. More general instructions for various different systems are available from the Haskell.org downloads page.

2.1.1 Installation from source with `cabal-install`

`ghc` 8.0.2 and **`cabal-install`** 2.0 or later may be available using your system package manager. Note that some distributions package older versions of **`ghc`** and **`cabal-install`** by default, so you may need to upgrade.

You can check the versions that are currently installed (if any) by running:

```
ghc --version
cabal --version
```

If newer versions are required, the easiest way to upgrade is to find an alternative package source that includes the newer versions. For Ubuntu 16.04 or later, and compatible distributions, a private package archive (PPA) is available. The following commands (run as `root` or with **`sudo`**) will register the PPA and install all the necessary dependencies:

```
add-apt-repository ppa:hvr/ghc
apt-get update
apt-get install cabal-install-2.0 ghc-8.0.2 libglpk-dev libblas-dev liblapack-dev
```

You will also need to add `/opt/ghc/bin` to your `PATH`, either by running `export PATH=/opt/ghc/bin:$PATH` or more permanently (e.g. by adding `PATH=/opt/ghc/bin:$PATH` to your `~/.profile` and running `source ~/.profile`).

For Debian, binary packages are available, and you should follow the instructions on that page. For other distributions, you should refer to the Haskell.org Linux downloads page to see if packages for your distribution are available.

If a pre-packaged version of **`ghc`** 8.0.2 is not available for your distribution, you can download binary packages from the GHC website.

If a pre-packaged version of **cabal-install** 2.0 or later is not available for your distribution, you can download binary packages from the Cabal website, or install an earlier version and upgrade by running:

```
cabal update
cabal install Cabal cabal-install
```

Once you have the dependencies installed, the following commands will check out the most recent version of the source code and build it:

```
git clone git@gitlab.com:well-typed/product-mix-auction.git
cd product-mix-auction
git submodule update --init

cabal update
cabal new-build
```

This will automatically download and compile Haskell library dependencies, if necessary.

You can then run `deploy/install.sh` (as `root`), which will install the built binaries under `/opt/pma` and install a `pma-server.service` unit file (assuming your host uses the `systemd` init system). If you wish to run the server or CLI app directly, add `/opt/pma/bin` to your `PATH`.

To download and build a more recent version of the sources, do the following:

```
git pull
git submodule update
cabal update
cabal new-build
```

You can then run `deploy/install.sh` (as `root`) again to install the updated binaries (you may need to stop the service and restart it after copying across the new files).

2.1.2 Installation from source with `stack`

stack 1.6.1 or later can be installed following the instructions in its documentation.

Once you have the dependencies installed, the following commands will check out the most recent version of the source code and build it:

```
git clone git@gitlab.com:well-typed/product-mix-auction.git
cd product-mix-auction
git submodule update --init

stack install
```

This will automatically download **ghc** and use it to compile Haskell library dependencies, if necessary. The **pma** and **pma-server** binaries will be installed in `~/.local/bin` by default, which you may need to add to your `PATH`.

To download and build a more recent version of the sources, do the following:

```
git pull
git submodule update
stack install
```

2.1.3 Building the documentation

The users' guide is written using the Sphinx documentation generator. If you wish to build the users' guide, install Sphinx following its installation instructions (for Ubuntu or Debian systems, you can `sudo apt-get install python-sphinx`).

Run `make <format>` in the `doc/` directory to build the docs in the given format. To see a list of supported formats, run `make` without any arguments. For example, to build and install the HTML format documentation:

```
cd doc
make html
cp -r _build/html /opt/pma/doc/html
```

To build the Haddock documentation, build the Haskell program as described above, then run the following in the `product-mix-auction` directory if you are using **`cabal`**:

```
cabal new-haddock
mkdir -p /opt/pma/doc
cp -r dist-newstyle/build/x86_64-linux/ghc-8.0.2/product-mix-auction-0.1.0.0/doc/
↪html/product-mix-auction /opt/pma/doc/haddock
```

Alternatively, if you are using **`stack`**:

```
stack haddock
```

2.2 Installation on Windows

The following instructions assume a 64-bit Windows edition. Building and installing on 32-bit editions should be possible, but will require a few adjustments and is not officially supported.

Moreover, the instructions explicitly specify versions of the dependencies that are known to work. It may be possible to use other versions, but this is untested and is likely to require more effort (especially if using a different version of **`MSYS`**).

If you wish to build a Windows installer from a Linux host using a virtual machine, see *Configuring a virtual machine* below.

2.2.1 Installation from source

1. Install **`stack`** 1.7.1 by downloading and running stack-1.7.1-windows-x86_64-installer.exe from the Stack website.
2. Download and unpack the **`Product-Mix Auction`** source code somewhere suitable (hereafter, the location of this directory will be referred to as `product-mix-auction`).

   If you wish to build the development version, it is recommended to use **`Git for Windows Portable`** (PortableGit-2.18.0-64-bit.7z.exe or a later release from the Git download page). This will avoid potential conflicts between different installations of **`MSYS`** from **`stack`** and **`git`**. Assuming you install this to `C:\PortableGit`, you can check out the source code by opening **`cmd.exe`** and running:

   ```
   C:\PortableGit\cmd\git.exe clone git@gitlab.com:well-typed/product-mix-auction.
   ↪git product-mix-auction
   cd product-mix-auction
   C:\PortableGit\cmd\git.exe submodule update --init
   ```

   Alternatively, you can download a ZIP file of the source code, or if installing in a VM with a Windows guest, you can clone the source code repository on the host and share the folder with the guest.
3. Download and unpack winglpk-4.63.zip into `product-mix-auction\thirdparty\glpk-4.63`. This contains pre-built Windows binaries from the WinGLPK project. Alternatively, you can try building from source yourself using the GLPK sources.
4. Download and unpack OpenBLAS-v0.2.19-Win64-int32.zip into `product-mix-auction\thirdparty\OpenBLAS-v0.2.19-Win64-int32`. This contains pre-built Windows binaries from the OpenBLAS project. Alternatively, you can try building from source yourself using the OpenBLAS sources.

5. Run **cmd.exe** and **cd** to the `product-mix-auction` source code directory.
6. Run **stack** which will install **GHC** and a suitable **MSYS** environment, and download necessary files:

```
stack --stack-yaml stack-win64.yaml setup
stack --stack-yaml stack-win64.yaml update
```

   If you receive an error message saying "certificate has unknown CA", look for the host name mentioned in the error message, and connect to it over HTTPS using Internet Explorer or Edge. This will typically be https://raw.githubusercontent.com. See the relevant Stack issue for more details.

   The second command may take some time to download the Hackage package index, but try not to interrupt it. If any downloads are interrupted, and re-trying the command fails, you may need to delete the partially downloaded files and start again. These are typically stored in `C:\Users\User\AppData\Local\Programs\stack\x86_64-windows` (or `stack path --programs`) and `C:\sr\indices` (under `stack path --stack-root`).

7. Start an **MINGW64** shell by running:

```
stack --stack-yaml stack-win64.yaml exec bash
```

   Note that there are multiple shells available with **MSYS**, and the behaviour of both **stack** and **MSYS** has changed over time. Check that `echo $MSYSTEM` returns `MINGW64` and `gcc -v` says that you are running **GCC** version 6.2.0 (distributed with **GHC** 8.2).

8. In the **MINGW64** shell, install the libraries needed for **openblas** using **pacman** (bundled with **MSYS**):

```
pacman -S mingw-w64-x86_64-gcc-fortran=7.3.0
```

   Note that you must use `mingw-w64-x86_64-gcc-fortran`, not `gcc-fortran`. Sometimes **pacman** package downloads are unreliable, so you may need multiple repeated attempts, but it should work out eventually. *Do not attempt to upgrade any pacman packages though, especially not pacman itself, because this may brick the MSYS environment.*

9. Copy the following libraries into the `product-mix-auction` directory (replacing `STACK_PROGRAMS_PATH` with the path given by `stack path --programs`, which will typically be `C:\Users\User\AppData\Local\Programs\stack\x86_64-windows`):

   - `thirdparty\glpk-4.63\w64\glpk_4_63.dll`
   - `thirdparty\OpenBLAS-v0.2.19-Win64-int32\bin\libopenblas.dll`
   - `STACK_PROGRAMS_PATH\msys2-20180531\mingw64\bin\libgcc_s_seh-1.dll`
   - `STACK_PROGRAMS_PATH\msys2-20180531\mingw64\bin\libgfortran-4.dll`
   - `STACK_PROGRAMS_PATH\msys2-20180531\mingw64\bin\libquadmath-0.dll`
   - `STACK_PROGRAMS_PATH\msys2-20180531\mingw64\bin\libwinpthread-1.dll`

   In the `product-mix-auction` directory, make a copy of `glpk_4_63.dll` named `glpk.dll` (so that **glpk-hs** can find it) and copy `libgfortran-4.dll` to `libgfortran.dll` and `libgfortran-3.dll` (so that **hmatrix** and **openblas** can find it).

10. Exit the **MINGW64** shell if necessary and return to the **cmd.exe** shell in the `product-mix-auction` directory. Finally, compile the package along with its Haskell dependencies:

```
stack --stack-yaml stack-win64.yaml install
```

    This will install `pma.exe` and `pma-server.exe` in the current directory (controlled by the `local-bin-path` option in `stack-win64.yaml`). You may wish to add this directory to your `PATH`. Alternatively, read on to *Creating an installer bundle*.

2.2.2 Creating an installer bundle

1. Follow the steps in *Installation from source* above to build `pma.exe` and `pma-server.exe`.
2. Install **`Inno Setup`** by downloading and running innosetup-5.5.9.exe from the Inno Setup website (later versions are also likely to work).
3. Open `deploy/win64/pma-win64.iss` with **`Inno Setup Compiler`**, and select *Build* → *Compile* from the menu.
4. **`Inno Setup Compiler`** should produce a self-contained Windows installer binary in `deploy/win64/output/pma-setup.exe`. Running this installer will (by default) install the application to `C:\Program Files\ProductMixAuction` and modify the `PATH` so that **`pma`** and **`pma-server`** are available.

2.2.3 Configuring a virtual machine

These instructions are necessary only if you do not have a Windows machine available, and wish to build the Windows installer using a virtual machine on a Linux host.

1. Download the MS Edge on Win10 image for VirtualBox and unzip it.
2. Install VirtualBox and its guest additions CD image. On Ubuntu-based systems, the required packages are `virtualbox` and `virtualbox-guest-additions-iso`.
3. Add your user account to the `vboxusers` group (`sudo usermod -a -G vboxusers $(whoami)`) and either log out and log in again, or run `su - $(whoami)` to give a shell with the new group membership available.
4. Run **`virtualbox`** and import the appliance (`.ova` file) downloaded earlier.
5. Configure the appliance settings. The ideal settings may depend on your exact system configuration, but the following example may be useful:
   - General: bidirectional shared clipboard
   - System: 8192 MB base memory, 4 processors, 90% execution cap, KVM paravirtualization
   - Storage: add an optical drive
   - Network: adapter 1 attached to Bridged Adapter
   - Shared folders: add a directory in which to build the software
6. Start the virtual machine. Insert the VirtualBox guest additions CD image (from the *Devices* menu), install the guest additions, then reboot the virtual machine.
7. If you are using **`Git for Windows`** to download the source code, and need access to non-public repositories, copy your SSH key to `C:\Users\IEUser\.ssh`.
8. Inside the virtual machine, follow the instructions from *Installation from source* above.

2.3 Installation on Mac OS X

2.3.1 Installation from source

For installation from source, a working Haskell toolchain is required. The easiest way to achieve this is to install stack. Stack requires **`Xcode`**, so install that first, then proceed to the stack installation.

The Product-Mix Auction package depends on GLPK. Download it from here, and follow the installation instructions contained in the package. Thus, something like:

```
curl -o glpk-4.63.tar.gz http://ftp.gnu.org/gnu/glpk/glpk-4.63.tar.gz
tar xf glpk-4.63.tar.gz
cd glpk-4.63
./configure
make
sudo make install
```

Once you have the dependencies installed, the following commands will check out the most recent version of the source code and build it:

```
git clone git@gitlab.com:well-typed/product-mix-auction.git
cd product-mix-auction
git submodule update --init
stack setup # only required the first time; this will also install a Haskell
            # compiler, so expect it to take a while
stack install --extra-lib-dirs=/usr/local/lib --extra-include-dirs=/usr/local/
↪include
```

This should give you two binaries under `~/.local/bin/`: **pma**, and **pma-server**.

2.3.2 Creating an installer bundle

This requires an additional dependency: **dylibbundler**. Installing from source:

```
git clone https://github.com/auriamg/macdylibbundler/
cd macdylibbundler
make
sudo make install
```

Creating app bundles: just run the `deploy/make-dmg.sh` script:

```
./deploy/make-dmg.sh
```

This will create DMG images for **pma** and **pma-server** in `./dist/osx`.

2.4 Getting started

Once you have the **pma** binary installed and available on your `PATH`, you can proceed to *Running auctions via the command-line interface*.

Once you have the **pma-server** binary installed and running, you can proceed to *Running auctions via the web app*. Note that the server requires the **pma** binary to be available on the `PATH`, or for the *`--pma-binary`* option to be used to specify its location.

CHAPTER

THREE

RUNNING AUCTIONS VIA THE COMMAND-LINE INTERFACE

The command-line program **pma** takes a complete specification of an auction (given via command-line parameters and CSV files), runs the auction and outputs the results (as CSV files and HTML pages containing graphs). This section explains how to run the program and interpret the output. It assumes that you are comfortable running programs via the command line.

Recall that a CSV (comma-separated value) file is essentially a table of data written in plain text format separated by commas and line breaks; it can be produced by spreadsheet programs such as Microsoft Excel. In this section, CSV files are represented as tables of data, with a source link in the caption that provides the original file.

Note that it is outside the scope of the package to specify how bid data ends up on the computer running the program. For a real-world auction, bidders could perhaps be asked to submit their bids to the auctioneer in a CSV file using a secure mechanism such as PGP-signed email, and the auctioneer could then run the **pma** program on the CSV files with a pre-determined set of parameters.

3.1 Usage example

To run a simple auction, first make sure that the **pma** executable is installed and on your `PATH` (see *Installation*). Save the files `basic-bids.csv` and `supply.csv` to a directory your local disk, then run the following command from the same directory:

```
pma lp --supply-file supply.csv --bids-file basic-bids.csv
```

This will run an auction and print the results to standard output. The following subsections describe the available command-line options and the format of the CSV files in detail.

3.2 Command-line parameters

The **pma** program supports sub-commands for the different kinds of auction:

- `pma lp` runs a standard Product-Mix Auction using the linear programming based solver (described below);
- `pma bc` runs a custom solver for auctions with budget-constrained bids (see *Budget constraints*);
- `pma dot-bids` runs a custom solver for auctions with positive and negative dot-bids (see *Positive and negative dot-bids*);
- `pma json` invokes the machine-readable interface (see *JSON interface*).

Use *`--help`* to see a brief summary of command-line parameters supported by the program.

The minimum information that must be specified to run an auction is a supply of goods (using *`--supply-file`*) and a list of bids (using *`--bids-file`*). These files must be formatted as described below in *Input format for supply* and *Input format for bids*.

`--help`
: Displays brief help on using the command-line program.

--supply-file CSV-FILE
Specify the file containing the supply curve information.

--bids-file CSV-FILE
Specify a file containing bids.

The *--bids-file* option may be used multiple times, in case each bidder has submitted bids in a separate file.

3.3 Input format for bids

Bids are entered in CSV files with one bid per row. The columns of the CSV file depend on which variations are in use. The default is for basic bids, but generalised or asymmetric bids may be specified using options:

--generalised-bids
Read the input bids as generalised (with max quantity columns for each good).

--asymmetric-bids
Read the input bids as asymmetric (with trade-off columns for each good).

For the basic auction (without the *--generalised-bids* or *--asymmetric-bids* options), the bids CSV file must have three fixed columns:

- A string label that uniquely identifies each bidder.
- A string label that identifies the individual bid.
- A bid quantity (as a positive integer).

These must be followed by one price column for each good in the auction. Prices must be integers, or may be omitted, in which case a value of 0 will be assumed.

The CSV file must have a header row, but the text of the headers will be ignored (so in particular, it is not correct to permute the order of columns).

Table 3.1: `--bids-file` basic example

Bidder	Bid	Quantity	Price for good 1	Price for good 2
A	1	5	120	75
B	1	1	100	
B	2	10	0	200

In this example, bidder A has bid for a maximum of 5 units at a price of up to 120 for good 1 or 75 for good 2, and bidder B has submitted two bids.

The association of bids with bidders does not affect the running of the basic auction, but makes a difference to the presentation of the output, and is relevant when additional constraints are in use (see *Additional constraint options* below).

3.3.1 Generalised bids

If the *--generalised-bids* option is in use, each price column must be preceded by a maximum quantity column for the corresponding good. Each maximum quantity must be a nonnegative integer, or may be omitted, in which case a value of 0 will be assumed.

As with basic bids, the third column must contain a limit on the quantity across all the goods.

For example:

Table 3.2: `--bids-file` example with generalised bids

Bidder	Bid	Overall quantity	Max quantity for good 1	Price for good 1	Max quantity for good 2	Price for good 2
A	1	5	5	120	3	75
B	1	1	1	100		
B	2	12	10	0	5	200

In this example, bidder A has bid for a maximum of 5 units, consisting of up to 5 units of good 1 at a price of 120 or up to 3 units of good 2 at a price of 75.

3.3.2 Asymmetric bids

If the *`--asymmetric-bids`* option is in use, each (maximum quantity column and) price column must be preceded by a trade-off column for the corresponding good. This must be a positive integer.

For example:

Table 3.3: `--bids-file` example with asymmetric bids

Bidder	Bid	Overall quantity	Trade-off for good 1	Price for good 1	Trade-off for good 2	Price for good 2
A	1	6	2	120	3	75
B	1	1	1	100		
B	2	12	1	0	3	200

In this example, bidder A has bid for up to 6/2 units of good 1 at per-unit price 120, or up to 6/3 units of good 2 at per-unit price 75.

When both the *`--generalised-bids`* and *`--asymmetric-bids`* options are in use, the trade-off column comes first, as shown in this example:

Table 3.4: `--bids-file` example with generalised asymmetric bids

Bidder	Bid	Overall quantity	Trade-off for good 1	Max quantity for good 1	Price for good 1	Trade-off for good 2	Max quantity for good 2	Price for good 2
A	1	6	2	4	120	1	3	75
B	1	1	1	1	100			
B	2	12	1	12	0	3	6	200

In this example, bidder A has bid for a maximum of 6 units divided by the relevant trade-off value, consisting of up to 4/2 units of good 1 at price 120, or up to 3/1 units of good 2 at price 75. If the auction determines prices of 100 for good 1 and 10 for good 2, the bid will receive all 3 units of good 2, then the overall maximum limits the number of good 1 units received to 3/2.

3.4 Input format for supply

See the *Supply* subsection for discussion of supply curves in general. The supply ordering can be chosen using command-line options:

`--horizontal-supply`
: Interpret supply curves as forming a "horizontal" supply, where each good is priced relative to selling nothing.

`--vertical-supply`
: Interpret supply curves as forming a "vertical" supply, where each good is priced relative to the preceding good.

--tabular-supply ROWS
: Make a tabular supply with the given number of goods per column (using a ragged final column if there are not enough goods). Using 1 good per column gives a horizontal supply, and using a number larger than the number of goods gives a vertical supply.

--tabular-supply-with-base ROWS
: Make a tabular supply with the first good below every column, and the given number of goods per column (using a ragged final column if there are not enough goods). Using a number larger than the number of goods gives a vertical supply.

The *--supply-file* option can be used to provide a CSV file from which supply curve data is taken. The first two columns contain the quantity (width of step) and price (height of step) for the first good, and so on for subsequent goods.

Table 3.5: `--supply-file` example

Quantity of good 1	Price for good 1	Quantity of good 2	Price for good 2
4	0	6	10
2	5	0	0

In this example, 4 units are available of good 1 with no reserve price, and a further 2 units are available provided a reserve price of 5 is met. Only 6 units of good 2 are available and with a reserve price of 10 (above the price of good 1, assuming *--vertical-supply* is used).

Note that it is permissible to have a step width of 0, in which case the step will be ignored, and that step widths (quantities) are not cumulative.

3.5 Input format for TQSS

See the *Total Quantity Supply Schedule* subsection for an explanation of the general idea of a TQSS. In order to use a TQSS, the *--tqss-file* option must be specified giving the TQSS supply function as a CSV file.

--tqss-file CSV-FILE
: Name of file in which the TQSS function specification can be found.

The CSV file describing the TQSS function must contain two columns, giving step widths and prices respectively.

Table 3.6: `--tqss-file` example

Step width	Mean price
5	0
1	80
1	112
0.5	140
0.5	160
1	200

TQSS prices may be absolute (see *TQSS with absolute prices*) or normalised (see *TQSS with normalised prices*):

--absolute-prices
: Use absolute prices in the definition of the TQSS (this is the default). Further options may be used as documented below.

--normalised-prices
: Use normalised prices in the definition of the TQSS. The *--horizontal-supply* option must be used. Further options are not available.

3.5.1 Options for TQSS with absolute prices

The following options are available only when the *--absolute-prices* option is used (or neither option is specified, in which case absolute prices will be used by default).

The default is for the TQSS to be based on the mean price of all goods (option *--mean-tqss*), but the *--single-good-tqss* option may be used to choose a single good instead. (Note that the text in the header row of the CSV file is not relevant.)

--single-good-tqss GOOD
Make a TQSS based on the price of a single good.

--mean-tqss
Make a TQSS based on the mean price of all the goods.

There are two possible approaches to varying the size of the auction during TQSS solving: adding a total quantity constraint to the linear program, or scaling the supply curves. The *--supply-constraint* option causes a total quantity constraint to be used. Alternatively, the *--supply-scale-lambda* option causes the supply curves to be scaled (with the parameter λ, which must be specified, controlling the proportion of scaling for non-base goods).

--supply-constraint
Rather than scaling the supply during TQSS solving, add a total quantity constraint to the linear program.

--supply-scale-lambda RATIO
Parameter to control scaling during TQSS solving. Base goods will be scaled in proportion to the new total size. Subsequent goods will be scaled depending on the given parameter lambda, which must be in the interval [0,1] (0 scales all goods in proportion to the new total size; 1 does not scale goods other than the base goods).

Solving the TQSS requires searching for the intersection between supply and demand curves, by calculating the demand at a fixed number of auction sizes. A search strategy may optionally be specified, or by default the *--combined-search* strategy will be used.

--combined-search
Evaluate the TQSS on a range of points across the entire interval (like *--linear-search-all*). If an exact intersection does not lie on one of these points, perform an additional *--binary-search* to give a more precise result.

--binary-search
Perform binary search on the interval. (Recommended for the fastest results, where there is a unique solution.)

--linear-search-all
Evaluate the TQSS on a range of points across the entire interval. Choose the point closest to the intersection.

--linear-search-below
Perform linear search from below (terminating when a solution is reached).

In addition, the search step size and lower and upper bounds may be specified. By default, the step size will be chosen so that a fixed number of points are used (except for binary search, where this option is ignored). The default lower bound will be 0 when the *--supply-constraint* option is used, or the initial auction size (from the supply curves) when the *--supply-scale-lambda* option is used (as supply curves should normally be scaled upwards but not downwards). The default upper bound will be the quantity given by the final TQSS step (i.e. the upper limit of its domain). Note that with the *--supply-scale-lambda* option, the upper limit of the TQSS domain must be greater than the initial auction size from the supply curves, otherwise an error will result.

--tqss-step-size UNITS
Step size for TQSS search modes other than binary search.

--tqss-from UNITS
Lower bound of interval for TQSS search.

--tqss-to UNITS
Upper bound of interval for TQSS search.

3.6 Additional constraint options

When there are multiple bidders, additional constraints may be imposed on the amounts that they receive in the auction. These may be specified as an absolute maximum (a quantity independent of the size of the auction) or as a fraction of the auction size.

--bidder-absolute-max UNITS
Limit the maximum quantity received by each bidder to an absolute number of units.

--bidder-relative-max RATIO
Limit the maximum quantity received by each bidder to a fraction of the total size of the auction.

3.7 Rationing and rounding options

See *Rationing and rounding* for background on rationing.

--no-rationing
Let the linear programme do what it will do, and don't do anything about it.

--linear-demand STEPS
Tweak marginal bids to approximate a linear demand around the bid price, then re-run the auction to find adjusted allocations.

--linear-demand-prefer-paired-bids STEPS
Same as *--linear-demand* but apply the tweaks only to paired bids that are marginal on multiple goods, and subsequently reallocate to treat equally bids that are marginal on single goods.

For the *--linear-demand* and *--linear-demand-prefer-paired-bids* options, a positive integer number of steps to use for approximating linear demand may be specified. Increasing this number reduces arbitrariness in the rationing results, but also increases the computational cost of rationing. If the value 0 is used, the program will automatically choose a number of steps sufficiently large to ensure results are accurate to within the precision with which quantities are reported (controlled by *--scale-factor*). However, this number of steps may be large and hence incur a significant runtime overhead.

The default rationing method is to use *--linear-demand-prefer-paired-bids* with an automatically-chosen number of steps.

--scale-factor INT
Specifies the precision with which quantity results should be reported, as a number of decimal places. The default is 1 decimal place. Increasing this will increase the number of rationing steps used by default (see *Rationing and rounding options*) and reduce the magnitude of the tweaks to ensure unique allocations.

3.8 Maximisation strategy options

See *Maximisation strategies* for background on maximisation strategies. The maximisation strategy may be chosen explicitly using one of the following options:

--max-efficiency
Maximise efficiency of auction as a whole

--max-profit
Maximise auctioneer's profit

If unspecified, the *--max-efficiency* option is the default.

When the *--max-profit* option is used, the auctioneer's profit will be printed to the file specified by *--results-file* (or to standard output, if a file is not specified).

3.9 Output formats

The output data produced by running the auction can be directed to files using the following options, otherwise it will be printed to standard output by default.

--prices-file CSV-FILE
: For each good, the price determined by the auction and the total quantity allocated.

--allocs-file CSV-FILE
: For each bidder, the quantities of each good it is allocated.

--bid-allocs-file CSV-FILE
: For each bid, the quantities of each good it is allocated.

--tqss-points-file CSV-FILE
: For each TQSS point, the demand, the price and the supply.

--results-file TXT-FILE
: Results of the auction that are not specific to goods or bidders.

The *--prices-file* option produces a CSV file containing a header column and one column for each good, with rows for the auction price, lowest winning bid price and total number of units allocated. If no units of a good are allocated, the lowest winning bid price is taken to be the (first) reserve price from the supply curve.

Table 3.7: `--prices-file` example

	Good 1	Good 2
Auction price	50	125
Lowest winning bid price	75	150
Allocation	2.4	3.6

The *--allocs-file* option produces a CSV file containing one row for each bidder, with columns for the bidder label and the quantity allocated of each good. Only successful bidders (i.e. those that receive more than zero units) are included in the table.

Table 3.8: `--allocs-file` example

Bidder	Quantity of good 1	Quantity of good 2
1	1.0	0.0
2	1.0	0.0
3	0.0	2.0
5	0.4	0.6
6	0.0	1.0

The *--bid-allocs-file* option produces a CSV file containing one row for each bid, with columns for the bidder label, bid label and the quantity allocated of each good. Frequently the allocation will be zero for all goods except one, but this need not always be the case. Only successful bids (i.e. those that receive more than zero units) are included in the table. This file is not printed to standard output by default, only if it is explicitly requested.

Table 3.9: `--bid-allocs-file` example

Bidder	Bid	Quantity of good 1	Quantity of good 2	Quantity of good 3
01	1	8.0	0.0	0.0
02	1	10.0	0.0	0.0
03	1	0.0	0.0	20.0
04	1	20.0	0.0	0.0
05	1	0.0	10.0	0.0
06	1	0.0	0.0	20.0
07	1	0.0	20.0	0.0
09	1	0.0	0.0	12.0

The *--tqss-points-file* option produces a CSV file containing a list of points evaluated when running the TQSS. It has no effect if a TQSS with absolute prices is not in use. The "Demand" column contains the total size of the auction that was run, the "Price" column contains the auction price measure used by the TQSS at that size, and the "Supply" column contains the value of the TQSS function at that price.

Table 3.10: `--tqss-points-file` example

Demand	Price	Supply
1.1	160.0	8.0
2.1	160.0	8.0
3.0	150.0	7.5
4.1	140.0	7.5
5.0	112.5	7.0
6.0	87.5	6.0

The *--results-file* option produces a text file containing individual values determined by the auction, where these do not relate to particular goods, bidders or TQSS points. This will include:

- When a TQSS is in use, the total quantity found by solving the TQSS.
- When a TQSS with *--normalised-prices* is in use, the normalised price across all goods.
- When the *--max-profit* option is in use, the auctioneer's profit.

If none of these are applicable, no file will be output, even if the *--results-file* option is specified.

3.10 Graphical output options

The *--graphics-file* option produces a HTML page containing embedded graphs. These graphs are described further in the section *Interpreting graphical auction results*. Various options are available to control the generated graphs:

--graphics-file HTML-FILE
: Writes a HTML page with graphs to the given file, overwriting it if it exists already.

--demand-curve CURVE
: Calculate supply and demand curves of the given type (see *Supply and demand for each good* for details of available curve types and the default behaviour). This option can be specified multiple times.

--dotsize DOUBLE
: Size for the bubbles in the bubble charts (default: 20.0)

Whether to show bid quantities may be selected using one of the following (mutually exclusive) options:

--graph-show-bid-quantity
: Show the quantity of each bid next to the bubble (default)

--graph-hide-bid-quantity
: Hide the quantity of each bid next to the bubble

How to colour the bids in the graphs may be selected using one of the following (mutually exclusive) options:

--graph-bid-unique-colors
Use a different color for each bid (default)

--graph-bid-simple-colors
Use only three colors for bids (accepted, rationed, rejected)

3.11 Miscellaneous user options

--shuffle-bids
Randomly shuffle the bids and additional constraints before running the auction. The bidder/bid labels will be replaced with integer labels in a random order, with bids from different bidders interspersed. If the program may favour arbitrary bids (e.g. because rationing is not in use), this makes it unpredictable which bids will be favoured. One could go further and break large bids into smaller bids to make the bids more homogeneous, but this is not currently implemented.

--preference-order
Preference order for allocating goods (permutation of the good labels in descending order of preference); by default, the highest-index good is most preferred. Goods omitted from the preference permutation will not be tweaked, so to disable price tweaks entirely, pass this flag with no arguments.

3.12 Development options

The following options are intended for developers working on the Product-Mix Auction library itself, or integrating it into a larger project.

--debug
Generate debugging log output and write out the linear programmes passed to the LP solver in CPLEX LP format.

3.13 Generating test data

Instead of specifying input data as CSV files using the *--supply-file* or *--bids-file* options, the *--arbitrary-supply* or *--arbitrary-bids* options can be used to generate random test data. These are primarily useful for development and testing purposes. Each of them comes with options to adjust the random data generated:

--arbitrary-supply
Generate an arbitrary supply function

--arbitrary-bids
Generate an arbitrary list of bids

--num-goods INT
Number of goods for random data generation

When the *--arbitrary-supply* option is in use, either a supply ordering option may be specified explicitly, or the program will arbitrary choose *--horizontal-supply* or *--vertical-supply*. In addition, the following options may be used to customize the generated supply curves:

--arbitrary-supply-min-units INT
Minimum amount of units used to draw random supply steps

--arbitrary-supply-max-units INT
Maximum amount of units used to draw random supply steps

--arbitrary-supply-min-price INT
Minimum price used to draw random supply steps

--arbitrary-supply-max-price INT
Maximum price used to draw random supply steps

--arbitrary-supply-min-steps INT
Minimum number of steps used to draw a random supply

--arbitrary-supply-max-steps INT
Maximum number of steps used to draw a random supply

When the *--arbitrary-bids* option is in use, the following options may be used to customize the generated bids:

--arbitrary-min-price INT
Minimum price used to draw random bids

--arbitrary-max-price INT
Maximum price used to draw random bids

--arbitrary-bid-min-units INT
Minimum amount of units used to draw random bids

--arbitrary-bid-max-units INT
Maximum amount of units used to draw random bids

--num-bidders INT
Number of bidders to generate

--num-bids INT
Number of bids to generate per bidder

--num-j-paired-bids J INT
Number of j-paired bids to generate per bidder

The *--num-bids* and *--num-j-paired-bids* options are cumulative for each bidder. For example, `--num-bidders 5 --num-bids 1 --num-j-paired-bids 1 6 --num-j-paired-bids 2 3` would generate 50 bids, with each of the 5 bidders having 1 bid with completely random prices (i.e. an N-paired bid where N is the number of goods in the auction), 6 single bids and 3 paired bids.

To see the generated data, you can use the *--dump-bids* or *--dump-supply* options to write the bids or supply to specified files. If desired, the *--no-run* option can be used to avoid actually running the auction.

--dump-bids CSV-FILE [OUTPUT]
Dump input bids to a CSV file (useful with *--arbitrary-bids*). Pass - to dump to standard output.

--dump-supply CSV-FILE [OUTPUT]
Dump input supply to a CSV file (useful with *--arbitrary-supply*). Pass - to dump to standard output.

--no-run
Do not actually run an auction, only read / generate inputs. Useful in combination with *--arbitrary-bids*, *--arbitrary-supply*, *--dump-bids* and/or *--dump-supply*.

3.14 JSON interface

The command-line interface supports the possibility to read and write files in JSON (JavaScript Object Notation) format. This provides a compact single-file representation of auctions, which is convenient for developing programs to communicate with the auction solver.

The JSON objects are described by the Swagger schema generated by the **pma-server** *--swagger-schema* option. This schema (or the Haskell source code) should be consulted for precise definitions of the types referenced below.

The following options can be used with the `pma lp`, `pma bc` and `pma dot-bids` commands:

--json-request-file JSON-FILE
: Read the auction input (request) from a JSON file. This option can be used in place of all the input options, and allows a JSON file produced by the web application to be executed directly by the CLI. The format of the file must correspond to the chosen command:

  - For `pma lp`, the format must be one of `SimpleAuctionInput`, `MediumAuctionInput` or `AuctionInput`.
  - For `pma bc`, the format must be `BCAuctionInput`.
  - For `pma dot-bids`, the format must be `DBAuctionInput`.

--json-request-output-file JSON-FILE
: Write the auction input (request) to a JSON file, in the format supported by *`--json-request-file`*. This is primarily intended to convert inputs available as CSV files into a format suitable for loading into the web application (if possible).

  For `pma lp` the simplest format consistent with the data will be chosen. Note that some invocations of `pma lp` may produce auctions that cannot be loaded into the web application, because they use features of the `AuctionInput` format that cannot be represented in the simpler formats used in the web application (see *Web app limitations*).

--json-response-file JSON-FILE
: Write the auction output (response) as a JSON file. The format will be `AuctionOutput`, `BCAuctionOutput` or `DBAuctionOutput` as appropriate.

In addition to the command-line interface described above, the **pma** program supports a machine interface based on JSON format. This interface is used by **pma-server** when it receives a request to run an auction, but may also be useful in other contexts where it is necessary to execute auctions programmatically.

To run an auction using this interface, execute `pma json`, write the auction specification as a JSON object to standard input, then read the response as a JSON object from standard output. For example, if you have a file `request.json` representing an auction, entering the command `pma json <request.json` at the command line will run the auction and print out the results in JSON format. The < operator causes the file to be supplied to the program over standard input.

The auction specification should be a JSON-encoding of one of the `...AuctionInput` types, and the result will be a list whose first element is one of the `...AuctionOutput` types, corresponding to the *`--json-request-file`* and *`--json-response-file`* options. These are the formats used by the web app, so it is possible to save a `request.json` file from the web app and then execute it in the command-line interface using `pma json <request.json`. See also *Integration between the web app and CLI*.

CHAPTER

FOUR

RUNNING AUCTIONS VIA THE WEB APP

The web app provides a convenient interface for running simple auctions, primarily for demonstration purposes. All auction data (supply information, bids and other parameters) is provided by a single user through their web browser. They can then press a button to see the results of the auction immediately, both in tabular and graphical forms. Auction data and results are not stored by the server.

4.1 Accessing the web app

The web app is available at http://pma.nuff.ox.ac.uk:3000/. Alternatively, follow the *Installation* instructions to install the software locally. Once the **`pma-server`** program is installed and running, open http://localhost:3000/ in your web browser.

You can select an input mode for the web app using the drop-down menu. The "Basic" and "Full-featured" options give different ways to enter bid data for standard LP auctions. The "Budget constrained" and "+/- dot bids" options allow entering *Budget constraints* and *Positive and negative dot-bids* respectively. Note that changing the selected input mode will reset the form and clear any data that has been entered.

The fields of the "Auction input" form are discussed in the corresponding sections below. When you have entered data for the auction you would like to run, press the "Run auction" button to send the data to the server. Once the server responds with the results, they will be displayed in the "Auction output" section, in both graphical and tabular forms. See *Interpreting graphical auction results* for more details on the graphs that may be displayed. If an error occurs, it will be displayed in the "Auction output" region instead.

Press the "Cancel" button if you wish to abort running an auction because it is taking too long.

Press the "Save" button to download an encoded representation of the auction data in JSON (JavaScript Object Notation). You can later press the "Upload" button and select a file to open, clearing any data that was previously entered and changing the input mode if necessary.

Press the "Download (Input and Results)" button to download a `.zip` file containing the input data (and output data, provided you have run the auction). This is formatted in CSV files suitable for use with the CLI (see *Running auctions via the command-line interface*). The downloaded `.zip` archive contains a `.json` file that can be extracted and loaded back into the web interface. See *Integration between the web app and CLI* for more details.

4.2 Supply specification in the web app

See *Supply* for general background on a supply specification.

The "Supply Ordering" drop-down menu allows you to choose whether the supply curves should be interpreted horizontally (all goods are independent) or vertically (goods are priced relative to the previous good, and quantities for earlier goods include the amount supplied to later goods). (This option applies only to LP auctions, not budget-constrained or dot-bids auctions, which are implicitly horizontal.)

The number of goods in the auction is determined by the number of supply curves in the "Supply curves for each good" section. You can add and remove supply curves using the "+ Supply curve for good" and "X Supply curve

for good" buttons. For each supply curve you can add and remove steps using the corresponding buttons. There must be at least one supply curve, and all supply curves must have at least one step.

For dot-bids auctions, only a single supply quantity and reserve price can be entered (as if the supply curve had only one step).

4.3 Bids in the web app

See *Bids* for general background on bidders and bids.

The "Bidders" section contains a list of bidders in the auction, each of which contains a collection of bids. (If a single bidder is entered, the results are presented as if each bid came from a separate bidder.) There must be at least one bidder, and all bidders must have at least one bid. The format for entering bids depends on the choice of input format.

In the "Basic" input format, a tabular representation of bids is shown, and only simple paired bids can be entered. The first column contains the quantity (number of units requested). Subsequent columns contain the bid prices on each good, ordered from left to right. There will be one column for each good in the supply.

In the "Full-featured" input format, a more complex representation of bids is shown, allowing asymmetric and generalised bids to be entered. The "Overall bid quantity" field gives the maximum number of units requested by the entire bid. This is followed by a table of "Bid values", which specify a good, trade-off (for asymmetric bids), maximum quantity for this good (for generalised bids) and a bid price. The trade-off should be 1 for symmetric bids, and the number of units should be the overall bid quantity for non-generalised bids. Each bid must include at least one bid value.

In the "Budget constrained" input format, only a simple table of bids is used, analogous to the "Basic" format for a single bidder, but with the first column containing the bid's budget instead of a quantity.

In the "+/- dot bids" input format, multiple bidders may be provided as in the "Basic" format. The quantity must be an integer, which may be positive or negative.

4.4 TQSS in the web app

See *Total Quantity Supply Schedule*.

The "TQSS" section of the form allows a Total Quantity Supply Schedule to be specified, for the "Basic" and "Full featured" input modes. The "TQSS enabled" drop-down menu must be set to show the other fields.

The "Price normalisation and scaling" drop-down determines whether absolute or normalised prices are used, and the approach used to scale the auction. For horizontal auctions, the available choices are:

- "Absolute - scale supply", corresponding to `--absolute-prices --supply-scale-lambda`;
- "Normalised - constrain supply", corresponding to `--normalised-prices --supply-constraint`.

For vertical auctions, only the "Absolute - scale supply" choice is available. Other combinations of the command-line options are not exposed in the web app, because they are experimental or not supported.

When absolute prices are in use:

- The "Price measure" drop-down determines the measure on which the TQSS prices are calculated. It gives a choice of "Average price of goods" (corresponding to `--mean-tqss`) or "Price of specified good" (corresponding to `--single-good-tqss`). In the latter case, the good in question can be selected using the drop-down "Good" menu.
- The auction size will be controlled by scaling the supply curves, as with the `--supply-scale-lambda` option. In the vertical case, the value of the λ parameter (between 0 and 1) can be given in the "Supply scale lambda" box. In the horizontal case, this parameter will always be 0.

- The "TQSS steps" table contains a list of steps, as in the CSV file accompanying `--tqss-file`. The first step in the table is filled in automatically to have width corresponding to the auction size and price zero, and it cannot be changed.

When normalised prices are in use:

- The "Price measure" and "Supply scale lambda" options are hidden.
- The auction size will be controlled by adding an extra constraint to the linear program, as with `--supply-constraint`.
- The first step in the "TQSS steps" table can be chosen freely.

In the web app, it is not possible to set the TQSS search strategy, lower and upper bounds or step size. A `--combined-search` with sensible defaults will be used to deliver accurate results and make it possible to plot a suitable graph. This corresponds to the default behaviour for the command-line interface described in *Input format for TQSS*.

4.5 Rationing in the web app

See *Rationing and rounding*.

The "Rationing and rounding options" section of the form makes it possible to override the standard rationing strategy, or change the precision with which quantities are reported. This may be useful if the standard strategy is too slow, or if more precise results are required. The section is collapsed by default, but can be expanded by clicking the arrow button. These options are available in the "Basic" and "Full-featured" input modes, but not for budget-constrained or +/- dot-bids auctions.

The "Standard" rationing mode corresponds to the `--linear-demand-prefer-paired-bids` option, and the "Standard without preferring paired bids" mode corresponds to the `--linear-demand` option. The latter is available only in the "Full-featured" input mode. In either of these cases, a number of steps may be set in the "Rationing step count" box, or the value 0 means a number of steps will be chosen automatically.

The "Disabled" mode (corresponding to the `--no-rationing` option) does not perform any rationing, so the reported allocations are given directly by the solution to the LP. This solution might not be unique, so the results depend on arbitrary implementation details of the solver.

The "Rounding of allocations" drop-down menu makes it possible to choose the number of decimal places to which allocated quantities are calculated, corresponding to the `--scale-factor` option. Fewer decimal places may improve performance, in particular when using rationing with the automatic step count, as fewer steps are required to yield sufficiently precise results.

4.6 Maximisation strategies in the web app

In the "Basic" input format, the "Maximisation strategy" section of the form (collapsed by default) provides a drop-down menu to choose the objective function. The available options are:

- Bidders' surplus + seller's profit (maximise efficiency)
- Seller's profit (maximise profitability)

See *Maximisation strategies* for more detail on these options. When "Seller's profit (maximise profitability)" is chosen, the supply ordering must be set to horizontal, and the TQSS must be disabled, otherwise an error will be reported. Moreover, the rationing options will be ignored when this option is used.

The other input formats do not offer a choice of maximisation strategy. The "Full-featured" and "+/- dot bids" formats always maximise efficiency, while the "Budget constrained" format always maximises profitability.

4.7 Web app limitations

Not all features of the command-line interface are available in the web application, in order to keep the latter as simple as possible. In particular:

- Not all supply orderings are supported (in particular the `--tabular-supply` and `--tabular-supply-with-base` options are not available).
- Limited TQSS options are provided (see *TQSS in the web app*).
- Bids are not randomly shuffled (i.e. the `--shuffle-bids` option is not used).
- Additional constraints (see *Additional constraint options*) are not available.
- It is not possible to customise the generated graphs (see *Graphical output options*); sensible defaults are used.
- It is not possible to adjust the auctioneer's preference order for allocating goods, so the highest-index good is always most preferred (the `--preference-order` option).

4.8 Integration between the web app and CLI

You may wish to set up an auction specification in the web app, then switch to the command-line **pma** program to actually run the auction (e.g. if bidders are providing their bids via CSV files, or if you wish to gain access to features not supported by the web app). The "Download (Input and Results)" button produces a `.zip` archive, which can be extracted using a standard tool. This archive includes a complete description of the auction as a collection of CSV files and a batch file `run_auction.bat`, which contains the command necessary to run the auction using the command-line interface. If you have run the auction using the web interface, the output data will also be included in the ZIP archive.

On Windows, after extracting the archive you can double-click the `run_auction.bat` file to run the auction locally, without using the command prompt. In this case, it may be useful to edit the batch file (e.g. using Notepad) to add a line containing the word `pause` by itself; this will make it possible to read any output printed by the program (e.g. error messages).

The ZIP archive contains a file `request.json`, which is a machine-readable description of the downloaded auction in JSON format, suitable for loading into the web app or passing to the command-line interface (see *JSON interface*).

If you have an auction specification designed for the command-line interface (i.e. a collection of CSV files and program options) and wish to load it into the web app, the `--json-request-output-file` option makes it possible to produce a suitable `request.json` file.

4.9 Server command-line options

The server accepts the following command-line options to adjust its behaviour.

--port ARG
: Port number on which the server runs (default: 3000)

--timeout ARG
: Request timeout in seconds (default: 30)

--pma-binary ARG
: Path to pma executable to start for each auction request (default: "pma")

The following options are likely useful only for developers:

--swagger-schema
: Instead of starting the server, print out the Swagger schema used for communication with the frontend

`--development`
: Switch to development mode (`ui.html` is loaded dynamically)

`--debug`
: Generate verbose debug output every time an auction is run

In addition, it may be useful to set GHC runtime system options. For example, the maximum heap size used by the server can be limited to 256Mb with `+RTS -M256M -RTS`, or the same limit can be imposed on the individual processes handling requests with `--pma-binary "pma +RTS -M256M -RTS"`.

CHAPTER

FIVE

INTERPRETING GRAPHICAL AUCTION RESULTS

This section describes the graphs produced by the software, using either the `--graphics-file` option of the command-line interface, or automatically using the web app.

When graphs are generated using the command-line interface, they are all output as a single web page (HTML file). You can open this page in a web browser to view the graphs. If they appear too wide or too narrow, resize the width of your web browser window and refresh the page. When hovering over a graph, a control bar will appear in the top right corner, with buttons to save the displayed plot as a PNG image, edit it using an online service, or adjust the view.

When graphs are generated in the web interface, they appear in the "Auction output" column under the price and allocation tables. Each graph has a collapse/expand button to make it possible to focus on the items of interest. Again, the width of the graphs will be fixed automatically based on the width of the web browser window. The control bar to edit graphs is not displayed in the web app, but the command-line version of the graphs can be accessed by downloading the auction results as a ZIP file and opening the included `graphs.html` file.

For most graphs, you can click and drag to zoom in, and double-click to zoom out again. Clicking on the legend will toggle the display of individual traces, or double-clicking will display a single trace.

Note that in some cases the order and colour of bids may be different between the web app and CLI. This arises because the web app always labels bids with integers (in the usual order), but the CLI permits the CSV file to have arbitrary textual labels (in alphabetical order). For example, in graphs generated by the CLI, a bid labelled "10" appears between "1" and "2".

5.1 Bids bubble chart (2D)

This graph is generated only when there are exactly 2 goods. The bids are shown as bubbles in price space (with the price of one good on each axis). Filled circles represent bids that are fully allocated (i.e. receive the entirety of their requested quantity); partially filled circles represent bids that receive some but not all of their requested quantity; open circles represent bids that receive nothing. The size of the bubbles is constant (but can be adjusted using the `--dotsize` option). Text labels next to the bubbles display the quantity allocated to the bid, unless the `--graph-hide-bid-quantity` flag is passed. Unsuccessful bids are not labelled.

A generalised bid is considered "fully allocated" if the overall bid quantity constraint is binding or if at least one individual good's maximum constraint is binding, taking account of the trade-off for asymmetric bids if appropriate. Additional constraints are not taken into account. This is tested after rounding, so in some circumstances we may show a bid as fully allocated that actually receives less than its demand in an exact analytic solution.

The prices resulting from the auction are plotted as horizontal and vertical black line segments. The rectangular region bounded by these line segments and the axes contains bids that are rejected. Originating from the intersection of the line segments, a diagonal line separates the regions of bids that receive each good.

5.2 Bids bubble chart (3D)

This graph is generated only when there are exactly 3 goods. It is a three-dimensional version of the bubble chart described above, where bids are represented as bubbles in price space. Again, the coordinates reflect the bid price

on each good, and the bubbles are filled and labelled based on the allocated quantity.

The black lines showing the auction prices remain, and in addition the plane segments separating unique demand regions are plotted in semi-transparent colours.

When viewing the graph in a web browser, the view can be manipulated using the mouse. Left-click and drag to rotate, right-click and drag to translate and middle-click and drag to zoom.

5.3 Allocation of bids to goods

This graph is generated for all auctions. Each column represents a good, with the y-axis showing prices. Black horizontal lines represent the auction prices. Each bid is plotted in the columns for goods it receives (or in one of the columns for a good it would receive if the prices were lower). The vertical position corresponds to the bid price. The area of the bubbles is constant, and horizontal position within the column is not significant (other than to visually separate bids at the same price). Bubbles are labelled with the quantity of the good actually allocated to each bid, or the quantity requested for unsuccessful bids. Where bids are partially allocated, the quantity allocated is shown as a fraction of the quantity requested for the particular good (taking account of asymmetric and generalised bids).

Non-paired bids are plotted in the column for the good on which they bid. Paired bids are plotted in the column for the good they won or were closest to winning (resolving ties arbitrarily in the latter case). If a paired bid is rationed (i.e. wins multiple goods), it is plotted in every column corresponding to a good it receives.

If a bid receives its entire requested allocation of the relevant good, it is plotted as a filled circle. If it receives part of its requested allocation, but not the full amount, it is plotted as a partially filled circle (an inner filled circle and an outer unfilled circle). If it receives nothing, it is plotted as an unfilled circle.

As with the bubble chart, the `--dotsize` and `--graph-hide-bid-quantity` options can be used to adjust the display. In addition, the `--graph-bid-simple-colors` option can be used to colour the bids depending on whether or not they are accepted, rather than by bidder.

5.4 Supply and demand for each good

Supply and demand curves can be generated for each good individually, depending on the configuration. The x-axis shows good prices (expressed relative to the price of the previous good, for vertical auctions, or absolute prices for horizontal auctions). The y-axis shows the total quantity allocated to the good and (for vertical auctions) any successive goods. The supply curve is plotted as a continuous line, and a vertical dashed line shows the quantity allocated by the auction, with a marker at the auction price.

Various different types of supply and demand curve graphs are supported, depending on the auction configuration (in particular, whether it uses a horizontal or vertical supply ordering, and whether a TQSS is in use). These are described in the following subsections.

The web application will always generate the default supply and demand curves. In the command-line interface, the default behaviour may be overridden and demand curve types to generate may be specified using the `--demand-curve` option, including a number of points if appropriate. This option may be repeated multiple times, to generate multiple different supply and demand curve graphs.

The subsections below discuss the different demand curve types available. The supply curve will be the same in each case, except that in the `UnadjustedDemandWithTQSS` case (see *Supply and demand for horizontal auctions with TQSS*) it will be increased by the value of the TQSS.

When an absolute TQSS that scales the supply is in use, the supply curves are drawn at the equilibrium value, i.e. they vary according to the TQSS (see *TQSS with absolute prices*).

5.4.1 Supply curve only

This consists of a supply curve only, with no demand curve. This is the default graph for auctions that are neither simply horizontal or vertical (e.g. a tabular supply). It can be requested in the command-line interface using

`--demand-curve NoDemand`.

5.4.2 Supply and demand for horizontal auctions

This computes the aggregate demand of all the bids on a good, ignoring bidding on other goods (so paired bids are counted for multiple goods). It does not require a number of points to be specified, or result in additional runs of the LP. Since the demand curve is in principle correct for all quantities, it is plotted as a continuous line. The width of the graph will be the width of the supply curve.

This is the default graph for horizontal auctions. It can be requested in the command-line interface using `--demand-curve UnadjustedDemand`.

5.4.3 Supply and demand for horizontal auctions with TQSS

This is similar to *Supply and demand for horizontal auctions*, wherein the demand curve aggregates all bids on a good, but with the supply curve increased vertically by adding the TQSS. The width of the graph will be limited to the width of the TQSS, which may be less than the width of the supply curve.

By default, this is generated in addition to the previous graph for horizontal auctions with a constraint-style TQSS (except that in the web app, it is generated in "Full-featured" mode but not "Basic" mode). It can be requested in the command-line interface using `--demand-curve UnadjustedDemandWithTQSS`.

5.4.4 Supply and demand for vertical auctions

This runs the LP multiple times for different total quantities, fixing the supply curve for the current good such that the desired quantity will be supplied (using a large negative price). Since the demand is known only at quantity points for which an LP run is performed, the curve is plotted as a discontinuous series of points.

In the graph for good *j* and higher, each of these points signifies the "demand" price spread that the auction would set if it were required to allocate the quantity shown on the horizontal axis to the set of goods $j,\ldots,n$. Note that a price set by the auction may not correspond to actual (inverse) demand, if the actual bidding would yield no sale on that good.

This is the default graph for vertical auctions. It can be requested in the command-line interface using `--demand-curve "AdjustedDemand <n>"` where `<n>` is a positive integer number of LP runs. This will yield a demand curve consisting of *n* evenly-spaced points.

5.4.5 Fixed total allocation

This is similar to *Supply and demand for vertical auctions*, but the supply curve for every good is fixed, and a price of zero is used for the height of the supply curve steps, rather than a large negative price. That is, to calculate a demand point, an LP run is performed with the current good having a supply quantity specified on the y-axis, and other goods having the supply quantities calculated in the original run of the auction.

This is a non-standard (experimental) graph that will not be displayed unless explicitly requested on the command-line with `--demand-curve "FixedTotalAllocation <n>"` where `<n>` is a positive integer number of LP runs.

5.4.6 Variable total allocation

This is similar to *Fixed total allocation* and *Supply and demand for vertical auctions*. The supply curve of the base good in a vertical auction is left unchanged, allowing the total quantity supplied to vary. Other supply curves are fixed based on the quantities calculated in the original run of the auction, as in *Fixed total allocation*.

This is a non-standard (experimental) graph that will not be displayed unless explicitly requested on the command-line with `--demand-curve "VariableTotalAllocation <n>"` where `<n>` is a positive integer number of LP runs.

5.5 TQSS and demand

This graph is generated if a TQSS (see *Total Quantity Supply Schedule*) is in use. The x-axis shows the total size of the auction (i.e. the amount of goods available to allocate, not necessarily the amount actually allocated). The y-axis shows the price measure used by the TQSS, depending on how the TQSS was configured. The TQSS function itself, as specified in the input, is plotted as a continuous line. The demand curve is calculated differently depending on whether the TQSS prices are absolute or normalised, as described in the following subsections.

5.5.1 TQSS with absolute prices

For a TQSS with absolute prices, the demand price is calculated for particular sizes of auction, and the corresponding points are plotted. Since the price is calculated at only a limited number of auction sizes, if the price decreases between adjacent points, then the size(s) at which it changes will not be known precisely. Such sections of the demand curve will be plotted with diagonal dashed lines to represent the fact that they represent an unknown decreasing step function.

The domain of points at which demand is calculated are determined by the TQSS search strategy, plus the `--tqss-step-size`, `--tqss-from` and `--tqss-to` options. See *Options for TQSS with absolute prices* for more details on these options. In the web application (and by default in the command-line interface), evenly-spaced points are plotted along with additional points around the intersection between supply and demand (the `--combined-search` option). When supply curves are being scaled (the `--supply-scale-lambda` option), the lower bound is the auction size (i.e. the sum of the supply curve lengths for horizontal auctions, or the length of the bottom supply curve for vertical auctions), which is also used as the width of the enforced TQSS step at height zero.

5.5.2 TQSS with normalised prices

For a TQSS with normalised prices, an alternative version of this graph is generated. In this case, the y-axis shows the normalised price (i.e. the common price of all goods above their corresponding supply curves at the supplied quantity). The demand curve is calculated directly from the bids, so it is a continuous line, rather than interpolating a fixed number of points. Paired bids are treated as demanding the good(s) which they receive in the final auction result, or which they would prefer to receive (for unsuccessful bids). Ties are resolved arbitrarily, so for quantities above the total actually allocated (i.e. right of the intersection), the graph is approximate.

CHAPTER

SIX

BUDGET CONSTRAINTS

A Product-Mix Auction with budget constraints is a variant of the basic problem (see *Introduction*), where instead of bids specifying a quantity of goods they wish to receive, each bid specifies a budget available to spend. As usual, the auction calculates prices for each good and each bid receives the goods it prefers at those prices.

For example, a bidder might ask to spend 10 on either (good A at price 5) or (good B at price 2). If the auction determines prices of 2 for good A and 2 for good B, the bid will receive 5 units of good A (exhausting its budget). If the auction determines prices of 5 for good A and 2 for good B, the bid may receive any combination of units of A and B that costs no more than 10 in total.

The following sections outline the algorithm. For a more detailed specification, see the accompanying document *"Specification for Implementing the Budget-Constrained Product-Mix Auction Solver"* on http://pma.nuff.ox.ac.uk.

6.1 The algorithm

The algorithm takes as input:

- a cost curve (step function) for each good available in the auction, giving the auctioneer's costs as a function of the number of units sold;
- a list of budget-constrained bids, which consist of a label, a budget (maximal amount that can be spent) and a vector of prices that the bidder offers for each good.

The goal of the algorithm is to find the (per-good) unit price vector that maximises the auctioneer's profit. The outputs of the algorithm are the price at which the auctioneer should sell each good, the total quantity of each type of goods sold, the profit made by the auctioneer and an assignment of quantities of goods to each bid.

6.1.1 Interpreting bids' demand

Given some candidate price vector (the auction prices) for the goods, the rules for deciding whether a bid wins, and the quantity of goods it may receive, are determined based on the ratios of the bid price to auction price for the different goods, as follows:

- A bid is a *winning bid* for a good if it offers a price greater than or equal to the candidate unit price for that good (equivalently, if the bid_price / auction_price ratio is at least 1).
- A winning bid is *non-marginal* if there is a unique highest bid_price / auction_price ratio that is strictly greater than 1 and is achieved by a single good. Such a bid will necessarily get all of its budget worth of that single good.
- A winning bid is *marginal in a set of goods* if the highest bid_price / auction_price ratio is strictly greater than 1, and is achieved by more than one good. Such a bid may receive any linear combination of the goods in which it is marginal, but the combined value of the goods will necessarily equal the bid's budget.
- A winning bid is *marginal in its budget and a set of goods* if there is at least one good (but possibly more) for which the bid_price / auction_price ratio is 1, and that it is the highest ratio for the given bid and candidate

unit prices. Such a bid may receive any linear combination of the goods in which it is marginal, with a combined value anywhere between 0 and the bid's budget.

- If a bid is not a winning bid for a good, it receives no units of that good.

Note that a bid's demand is not well defined if the price for any good is zero. Thus we reject any candidate price vector containing a zero price. Correspondingly, we require that for each good, the set of bids contains at least one bid with a non-zero price for that good.

6.1.2 Search for optimal prices

Since unit prices and quantities are arbitrarily divisible, we cannot just enumerate their possible values. Instead, we calculate a set of candidate price vectors (vertices of the unique demand regions generated by the bids) amongst which the optimal prices must lie. We then search for the most profitable bid-to-quantity assignment for each candidate price vector in this set, and return the prices that yield the greatest profit.

Two strategies are supported for choosing the set of candidate price vectors to search:

- Finding every intersection of the demand hyperplanes generated by the bids, which is potentially slow due to the large number of intersections (especially in auctions with more than a few goods), but should ensure the optimal value is found.
- A more refined algorithm for identifying a smaller set of candidate price vectors, which is likely to be faster but has not been proved correct.

For a single candidate price vector, we cannot consider all possible assignments of quantities to bids that satisfy the bids' demands, because marginal bids may receive arbitrary combinations of goods on which they are marginal. However, the structure of the cost curves means that it is enough to consider a finite set of possible assignments, amongst which the maximum profit must lie:

- Non-marginal winning bids always get their budget worth of a single good. If there is not enough supply to meet their demands, we can immediately reject the candidate price vector (knowing that there will be higher candidate prices at which there are fewer non-marginal winning bids).
- For marginal bids, we keep track of the current quantity sold of each good, and try all possible orders of assigning quantities of goods to bids.
- Bids that are marginal in the budget and some goods force us to explore several options for assigning some quantity of a good (on which the bid is marginal):
  - no units of the good;
  - enough units of the good to exhaust the remainder of the budget;
  - any positive quantity (whose value is less than the remaining budget) that, when added to the current quantity sold for that good, adds up to the quantity at the upper endpoint of a line of the cost curve for the good.
- Bids that are marginal in some goods (but not marginal in the budget) are similar, but once we have finished assigning goods to a bid, we make sure the assignments for that bid end up being worth the entire budget of that bid.

By exploring all the possible combinations of decisions that come out of this process, and only keeping the decisions that abide by the rules, we get all the candidate quantities and allocations. All that's left to do at this point is to look at the allocation that maximises the objective function (computing the auctioneer's profit). This gives us the best allocations for a given price vector. We have several candidate price vectors, so we just repeat this process for each price vector and keep the one that maximises the auctioneer's profit.

We return an arbitrary allocation of goods to bids that results in the maximum profit. There may well be multiple possible allocations that result in the same profit, and in this case we do not attempt any kind of fairness between bids.

6.2 Budget constraints in the command-line interface

The budget constrained bids functionality is available in the command-line interface under `pma bc` (see *Running auctions via the command-line interface*). To run a budget-constrained auction, the following options must be specified:

--supply-file `CSV-FILE`
Specify the file containing the cost curve information.

--bids-file `CSV-FILE`
Specify a file containing bids.

The following output file locations may optionally be specified, otherwise they will be written to standard output by default:

--prices-file `CSV-FILE`
For each good, the price determined by the auction and the total quantity allocated.

--allocs-file `CSV-FILE`
For each bidder, the quantities of each good it is allocated.

--results-file `TXT-FILE`
A text file containing the auctioneer's profit.

The set of candidate price vectors to search can be chosen using the *--all-prices* or *--filter-prices* option. By default, *--all-prices* will be used.

--all-prices
Explore all hyperplane intersections when determining candidate auction prices.

--filter-prices
Use the smarter heuristic for decreasing the number of candidate auction prices - NOT proved correct.

The cost curve data is specified using the *--supply-file* option to specify a CSV file in the same format as for the standard Product-Mix Auction, described in *Input format for supply*. The supply ordering is always horizontal. For example, here are cost curves for 3 goods, each with 2 steps, with a price of 1 for the first 10 units and a price of 2 for the next 30 units:

Table 6.1: `--supply-file` example

Quantity of good 1	Price for good 1	Quantity of good 2	Price for good 2	Quantity of good 3	Price for good 3
10	1	10	1	10	1
30	2	30	2	30	2

The *--bids-file* option is used to specify the bid data in a CSV file with columns for the bid label and total budget, then one column per good giving the bid price. This is rather like the standard format described in *Input format for bids*, except that the second column is a budget rather than a maximum quantity.

Table 6.2: `--bids-file` example

Bid	Budget	Price for good 1	Price for good 2	Price for good 3
A	20	2	3	5
B	31.2	2.2	2.8	4

The *--prices-file*, *--allocs-file* and *--results-file* options are used to output results to the named files. If not specified, results will be written to standard output (printed out to the console).

For example, running `pma bc` with the above cost curves and bids produces the following output files:

Table 6.3: `--prices-file` example

	Good 1	Good 2	Good 3
Auction price	2.2	2.8	4.0
Allocation	0.0	4.0	10.0

Table 6.4: `--allocs-file` example

Bid	Quantity of good 1	Quantity of good 2	Quantity of good 3
A	0.0	0.0	5.0
B	0.0	4.0	5.0

In the command-line application, input quantities and prices may be specified as decimals (internally they are represented as double-precision floating-point numbers). Output prices will not be rounded, but should not be assumed to be arbitrarily precise. Output quantities will be rounded based on the *`--scale-factor`* option.

`--scale-factor` `INT`
: Specifies the precision with which quantity results should be reported, as a number of decimal places. The default is 1 decimal place.

As in LP auctions, the *`--arbitrary-bids`* and *`--arbitrary-supply`* options can be used to generate random data instead of providing input files (see *Generating test data* for further discussion).

`--arbitrary-supply`
: Generate an arbitrary supply function

`--arbitrary-bids`
: Generate an arbitrary list of bids

`--num-goods` `INT`
: Number of goods for random data generation

`--arbitrary-supply-min-units` `INT`
: Minimum amount of units used to draw random supply steps

`--arbitrary-supply-max-units` `INT`
: Maximum amount of units used to draw random supply steps

`--arbitrary-supply-min-price` `INT`
: Minimum price used to draw random supply steps

`--arbitrary-supply-max-price` `INT`
: Maximum price used to draw random supply steps

`--arbitrary-supply-min-steps` `INT`
: Minimum number of steps used to draw a random supply

`--arbitrary-supply-max-steps` `INT`
: Maximum number of steps used to draw a random supply

`--arbitrary-min-price` `INT`
: Minimum price used to draw random bids

`--arbitrary-max-price` `INT`
: Maximum price used to draw random bids

`--arbitrary-bid-min-budget` `INT`
: Minimum budget used to draw random bids

`--arbitrary-bid-max-budget` `INT`
: Maximum budget used to draw random bids

`--num-bids` `INT`
: Number of bids to generate

--num-j-paired-bids J INT
Number of j-paired bids to generate

--dump-bids CSV-FILE [OUTPUT]
Dump input bids to a CSV file (useful with *--arbitrary-bids*). Pass - to dump to standard output.

--dump-supply CSV-FILE [OUTPUT]
Dump input supply to a CSV file (useful with *--arbitrary-supply*). Pass - to dump to standard output.

--no-run
Do not actually run an auction, only read / generate inputs. Useful in combination with *--arbitrary-bids*, *--arbitrary-supply*, *--dump-bids* and/or *--dump-supply*.

6.3 Budget constraints in the web application

The web application (see *Running auctions via the web app*) supports a "Budget constrained" input mode. This provides a graphical interface for specifying cost curves and budget-constrained bids. Running a budget-constrained auction will report the auctioneer's profit (the optimal value of the objective function) and display prices and allocations of goods to bids that yield the optimal value.

The web application requires prices to be entered as integers, and rounds output prices to the nearest integer (except the auctioneer's profit, which may be displayed more precisely). Calculated allocations are always rounded to at most 1 decimal place.

It is not currently possible to choose the set of candidate price vectors to search using the web application. The *--filter-prices* option, which uses a smaller set of candidates, is always used.

6.4 3D graph of budget constraints

For help visualising budget constraints in three dimensions, an `interactive 3D plot of budget constraints` is available. This renders the planes separating the unique demand regions arising from bids.

The parameters of the graph can be changed by opening the page in a web browser, then extending the URL query string with key-value pairs, for example replacing `budget-constraints-3d.html` with `budget-constraints-3d.html?x=10&y=20&z=30`. The following parameters can be changed:

- `mx`, `my`, `mz`: dimensions of the graph (maximum x, maximum y, maximum z);
- `x`, `y`, `z`: coordinates of the first bid;
- `x1`, `y1`, `z1` .. `x9`, `y9`, `z9`: optional coordinates of additional bid points;
- `opacity`: controls the transparency of the planes: a value between 0 and 1, where 0 is transparent and 1 is opaque;
- `budget`: the value `no` renders unique demand regions for bids in the standard Product-Mix Auction, rather than budget-constrained bids.

Coordinates of bids can be set to a number between 0 and the corresponding maximum value, or to the special value `r`, which will cause the coordinate to be chosen at random.

For usage examples, see the HTML version of this document.

Note that in some web browsers, it may be necessary to access the page from a `http[s]:` URL (i.e. a web server) rather than a `file:` URL (i.e. a file stored on the local disk).

CHAPTER

SEVEN

POSITIVE AND NEGATIVE DOT-BIDS

A Product-Mix Auction with positive and negative dot-bids is a variant of the basic problem (see *Introduction*), where quantities are indivisible (i.e. a bid for 1 unit must receive exactly 1 unit, not 0.5 units) and bids may request negative quantities.

7.1 Format of the Problem

An auction consists of:

- A set of *goods*; these can be anything, but the quantity auctioned, demanded, and sold on each of them will be integral (see above).
- A set of *bidders*; each bidder is identified by a string label. Any label is valid, the only constraint is that no two bidders may have the same label.
- A *reserve price* for each good. The algorithm will never sell any goods below the reserve price for that good; bids will not be considered for goods on which they fall below the reserve price.
- A *target bundle*, which is a set of available supply per good. For many types of auctions, the auctioneer has exactly one unit of each good available, so that is the default.
- For each bidder, a set of *bids*.

The combination of reserve price and target bundle is equivalent to a supply curve with exactly one step.

A bid consists of:

- one desired integral quantity; and
- the maximum price the bidder is willing to pay for each good.

This format allows a bid to express an *alternative choice*: a bidder can bid on a fixed quantity of any of a range of goods, specifying the minimum price they are willing to pay for each.

For example, let's assume we're auctioning off FM radio frequencies, and we have three of those on offer, let's call them A, B, and C. Now a bidder can put in a bid that says "I want to buy one frequency, I am willing to pay up to 6000 for frequency A, up to 3000 for frequency B, or up to 2000 for frequency C":

Table 7.1: Bids example

Bidder	Weight	A	B	C
AcmeCorp	1	6000	3000	2000

If a bidder is interested only in some of the goods, but not others, then this can be expressed by setting the prices on the uninteresting goods to some value below the reserve price. The algorithm will never sell below reserve price, so such bids will never be accepted on these goods.

It is also possible to use *negative weights* for bids. These bids are used to cancel positive bids, and enable us to depict *any* strong substitute preferences in our language. Negative bids can only be placed on price combinations for which the same bidder is guaranteed to accept sufficient goods from other, positively weighted, bids. (Note that a negative bid *cannot* be interpreted as a "bid to sell": a negative bid is accepted on a good only if the price of

that good is *below* the bid for that good, whereas a seller would wish to sell a good only if its price were *above* a certain level.)

In order for this to work, negative bids can only be placed on price combinations for which the same bidder is guaranteed to accept sufficient goods from other, positively weighted, bids. For example, the following combination would be valid:

Table 7.2: Bids example

Bidder	Weight	A	B
AcmeCorp	1	6000	3000
AcmeCorp	-1	5000	2000
AcmeCorp	1	5000	0
AcmeCorp	1	0	2000

7.2 The Working Of The Algorithm

The auction algorithm is a two-step process.

7.2.1 Step 1: Finding equilibrium prices

In order to find equilibrium prices, the algorithm performs a series of *Submodular Function Minimisation* steps.

Informally, this process finds the lowest possible prices for which the total demand is less than or equal to the target bundle on every good. In other words, we find the lowest possible prices at which there is no excess demand.

The algorithm is designed so that such prices will always exist: at the reserve price of each good, the seller has placed bids to "buy back" more than the entire supply of this good. So each price may drop to its reserve price, but if it drops lower, there will be excess demand.

7.2.2 Step 2: Allocating

The allocation algorithm is somewhat complex, but essentially, it consists of three steps that are applied iteratively until either all bids have been dealt with, or the entire supply has been sold.

The first step is the *obvious* step, in which all bids are dealt with that are unambiguous:

- If a bid is higher than the equilibrium price by a larger amount on one good than on all the others (i.e., it is *non-marginally accepted*), then the algorithm allocates on that good.
- If a bid is lower than the equilibrium price on all goods (*non-marginally rejected*), then the algorithm rejects it.

The second step is the *unravel* step, in which we look at groups of bids that are ambiguous (marginal), but such that there is no more than one good among them for which more than one bidder needs to be considered. For such a group, we can unambiguously figure out how much to allocate:

- On the goods for which only one bidder has any marginal bids, we can allocate the entire remaining supply of that good on that bidder.
- On the one good for which multiple bidders have marginal bids, we can calculate the maximum possible allocation based on what we've allocated on the other goods, and the remaining supply.

The third step is the jiggle step, in which we resolve ambiguous situations by picking a combination of bidder and good, and giving it a slight advantage by temporarily moving all the bids of this bidder up by half a unit on that good. We then find the new equilibrium prices as above. This may cause bids to either become non-marginal, or it will break ambiguous groups such that the unravel part can resolve them. So we perform the obvious and unravel steps again, and then move bids back to where they were before. This step, thus, acts as a tie breaker, and our choice of bidder and good determines who gets priority.

7.3 Using the CLI

The dot-bids auction functionality is available under the subcommand `pma dot-bids`. Available options are:

--bids-file CSV-FILE
: Specify a CSV file containing the bids. The first column is the bidder name, the second column is the bid weight, subsequent columns are interpreted as bid prices per good.

  The first row (CSV header row) should contain good labels in columns 3 and up, referring to the goods in that column. Columns 1 and 2 in the header row are ignored, but should, by convention, contain the labels "Bidder" and "Weight".

  Without this option, CSV data is read from standard input instead.

--supply QUANTITIES
: Available supply: space-separated list of quantities, one per good, in double quotes. If not specified, defaults to one unit of every good.

  The number of quantities in the list must match the number of goods in the auction, as determined by the number of columns in the bids file (see *--bids-file*), or *--num-goods* for the arbitrary-bids generator (see *--arbitrary-bids*).

--reserve-price PRICES
: Reserve prices: space-separated list of prices, one per good, in double quotes. If not specified, defaults to a reserve price of zero for every good. The auction algorithm will never sell anything below this price.

  The number of prices in the list must match the number of goods in the auction, as determined by the number of columns in the bids file (see *--bids-file*), or *--num-goods* for the arbitrary-bids generator (see *--arbitrary-bids*).

--supply-file CSV-FILE
: Supply of goods available in the auction, and their reserve prices, represented as a CSV file with a header row and exactly one data row. This may be used instead of both the *--supply* and *--reserve-price* options. (This is to match the format of other auction types, where multiple rows are used to represent multiple supply curve steps.)

  The number of goods in the file must match the number of goods in the auction, as determined by the number of columns in the bids file (see *--bids-file*), or *--num-goods* for the arbitrary-bids generator (see *--arbitrary-bids*).

--allocs-file CSV-FILE
: Output the resulting allocations to the given CSV file. The output format consists of a column for the bidder name, followed by one column per good giving the allocated quantities. The last row lists unsold supplies under the bidder name "UNSOLD".

--prices-file CSV-FILE
: Output the resulting equilibrium prices to the given CSV file. The output format consists of a header column and one column per good: the two data rows give the price for the good and the total quantity sold.

--log [FILE]
: Log the internal steps of the allocation algorithm to FILE, or to standard error if no file is specified.

--arbitrary-bids
: Generate a random set of arbitrary bids. Normally, this is fed to the auction resolver directly and silently; in order to view the generated auction, use *--dump-bids*, and to bypass the auction resolver and just run the auction generator, use *--no-run*.

--num-goods N
: Number of goods in an auction. This is mostly useful when using *--arbitrary-bids* (the *n* parameter), but can be specified as a sanity check in other cases. The default, 2 goods, is used only if the number of goods is not determined by any other inputs.

--max-weight WEIGHT
: Maximum weight (bid quantity) to generate. The default, 1, generates only unit bids. (default: 1)

--complexity B
: The b parameter. Higher values for b create more complex auctions. At $b = n$, bids are likely to collide a lot. (default: 3)

--dump-bids CSV-FILE [OUTPUT]
: Dump input bids to a CSV file (useful with *--arbitrary-bids*). Pass − to dump to standard output.

--dump-supply CSV-FILE [OUTPUT]
: Dump input supply to a CSV file. Pass − to dump to standard output.

--no-run
: Do not actually run an auction, only read / generate inputs. Useful in combination with *--arbitrary-bids* and *--dump-bids*.

INDEX

Symbols

A

B

R

S

T