MATHEMATICA FOR RESEARCHERS

PER ALEXANDERSSON

Abstract. Introduction to Mathematica for graduate students.

1. Introduction

1.1. **Becoming friends with Mathematica.** The most important feature in Mathematica is the F1 button, which brings you to the built-in documentation. Pressing F1 while the line cursor is on a function, brings you to the documentation about that particular function.

Notice that *all* Mathematica functions and options begin with a capital letter. For example, e, i and π are denoted `E, I` and `Pi`.

1.2. **Variables.** Variables are used for storing data. Variables can be indexed, and the index could be numbers, variables, strings or any data structure. Indexed variables are called *maps* in other programming languages.

```
x = 5;
myVar[1] = 4+2;
myVar[Pi] = 3.4;
myVar[2,5,6] = 63;
myVar[z] = 77;
myVar["first"] = 20;
```

If a value has not been assigned to a variable, the variable is considered to be undetermined or unknown. Unknowns are used when solving equations.

Once a value have been store in a variable, it can be "reset" to an undefined state by using either `variable =.` or `Clear[variable]`.

Notice that non-indexed variables which are undetermined appear in blue, while variables to which a value is assigned appear in black text.

Version: Monday 25th August, 2014,21:09.

1.3. **Functions and modules.** Every mathematician has an idea what a function is. In Mathematica, it is a way to assign values to a variable according to some rule, or pattern.

```
myFunction[x_] = x^2 + 2;
```

Special cases for certain values may be added: Defining special cases of functions can be done as follows:

```
sinc[0]:=1;
sinc[x_]:=Sin[x]/x;
```

Notice that without the special case, `sinc[0]` would be undetermined.

There are ways to add constraints on the input, so that rules only apply if certain conditions are met. For example, one can define functions that are only defined for integers.

Functions are mainly used to encapsulate code, so that it can be reused multiple times. Here is a definition of a function that takes two parameters:

```
myFunction[p_,q_]:=2p+q;
```

You may call the function with both numbers and symbols:

```
myFunction[5,6]
myFunction[a,b+c]
```

More complicated functions might need intermediate steps. These should be encapsulated by the `Module` command:

```
pqFormula[p_,q_]:=Module[{pHalf, inRoot,theSquareRoot},
    pHalf = p/2;
    inRoot = pHalf^2 - q;
    theSquareRoot = Sqrt[inRoot];
    Return[ -pHalf + theSquareRoot ];
];
```

Local variables should be used as much as possible, to prevent undesired side-effects. The `Return` statement is used to return the result from the module. To output intermediate results in a Module, use the `Print` function. This is a useful tool for finding errors.

1.4. **Set versus SetDelayed.** In Mathematica, the following code is literally interpreted as *Substitute all x on the right hand side, with the parameter provided, and then simplify.*

```
func[x_]:=...;
```

This is the `SetDelayed` way to define functions.

Using the `=` operator (`Set`) for assignment is on the other hand interpreted as *Simplify the right hand side once and for all,* then *substitute all x in the result when a parameter is provided.*

```
func[x_]=...;
```

Most of the time, there is no difference when the substitution occurs, but in some not too rare cases, there is:

```
funcDerivativeBad[x_]:=D[x^2, x];
funcDerivativeGood[x_]=D[x^2, x];
```

One can force Mathematica to evaluate sub-expressions, to make `:=` work:

```
funcDerivativeFixed[x_]:=Evaluate[D[x^2, x]];
```

As a rule of thumb, stick with `:=`, if you are uncertain. This is closest to the behavior in other programming languages.

1.5. **Calling functions.** There are various ways to call functions.

```
x = 2+3I;
Re[x]
Re@x
x//Re
```

You should be familiar with the first way already. The second alternative is similar to the mathematical $f \circ x$ notation, and the third one is called "piping". Piping has lower priority than `@`. These alternative forms are useful when applying multiple one-argument functions:

```
HankelMatrix[4] // Inverse // MatrixForm // Print;
Print @ MatrixForm @ Inverse @ HankelMatrix[4];
```

1.6. **Recursion, memoization.** Recursion is done as follows:

```
fib[0]:=0;
fib[1]:=1;
fib[n_]:=fib[n-1]+fib[n-2];
```

However, evaluating `fib[40]` takes a lot of time (`fib[1]` will be computed `fib[40]` times, and so on). One can rewrite the function to store intermediate results as follows:

```
fib[0]:=0;
fib[1]:=1;
fib[n_]:=fib[n] = fib[n-1]+fib[n-2];
```

Thus, each time we evaluate `fib[n]`, Mathematica redefines `fib[n]` to a number (the result of the computation). This is called *memoization*.

1.7. **Comments on coding practice.** Use comments that describe what and why longer functions and code blocks do:

```
(* This is a comment in mathematica *)

(*
Comments may span over
several lines, and can be used to temorarily remove code.
*)
```

Name your functions and variables, to reflect their use. For example, `ComplexListPlot` clearly states what it does.

2. Operations on data structures

2.1. **Creating lists.** Lists are the backbone in every automated process. These are used to store numbers, polynomials, functions, other lists etc. To construct a list, simply use one of the following:

```
List[1,2,Pi,a,b,c]
{1,2,Pi,a,b,c}
```

There are ways to automatically create lists where each element depend on one or more parameters:

```
Table[n^2, {n,1,10}] (* First 10 squares of natural numbers *)
Table[Binomial[n,k], {n,1,10},{k,1,n}] (* Pascal's triangle *)
```

Many built-in functions return lists.

It is sometimes useful to create a list of lists[1], or even more complex data structures. For example, the following might represent a simplicial complex:

```
{{1},{2},{3},{1,2},{1,3}}
```

The following is a feature that is good to be aware of: `{a,b}={2,3};` This assigns the values 2 and 3 to the variables `a` and `b` respectively.

[1]Matrices are represented as a list of lists (rows).

2.2. **Operations on lists.** In many situations, one wish to do something with every element in a list. To apply `f` on every element in `list`, simply use

```
Map[f, list]
```

For example, this piece of code factors a list of polynomials

```
polyList = Table[x^(2n)-1,{n,1,5}];
Map[Factor, polyList]

Factor /@ polyList (* Alternatively *)
```

2.3. **Anonymous functions.** One may create anonymous functions by using `#` as a placeholder for the argument, and `&` as the end of the definition. These are usually small functions.

```
#^2+4 &[5] (* This gives 29 *)
```

A common task is to split a list of complex numbers, into a list of pairs of real numbers, representing the real and imaginary part. By use of an anonymous function and `Map`, this can be done in the following way:

```
complexList = {1+2I, 3-I, I, 5};
pointList = {Re@#, Im@#}& /@ complexList;
```

2.4. **Functions returning new functions.** In some instances, it might be useful to create new functions. For example, Mathematica returns functions as solutions to differential equations, and functions that are used to solve various linear equations[2]. The following example creates a function that differentiates the argument with respect to x k times, and multiplies the result with `mulfunc`.

```
DifferentiateAndMultiply[k_, mulfunc_]:= mulfunc*D[#,{x,k}]]&;

(* Alternatively *)
DifferentiateAndMultiply[k_, mulfunc_]:=
Function[{f}, mulfunc*D[f,{x,k}]];
```

Example usage:

```
g = DifferentiateAndMultiply[2, x^2+2];
g[x^3] (* Returns 6x*(2+x^2) *)
```

[2]See `LinearSolveFunction`

3. Solving equations and making substitutions

3.1. **Solving equations.** Equations are expressions involving `==`, which is used as equality sign. This way, one may store equations in variables as follows:

```
myEquation = Sin[t] + 2*Cos[t] == Sin[2 t];
```

Solving equations is done by using `Solve` and `NSolve`:

```
Solve[x(x^2+4x+4)==0, x]
Solve[{x*y + 1 == 0, x + y == 5}, {x, y}]
```

`Solve` gives exact answers while `NSolve` solves the equations numerically. `Solve` and `NSolve` are used both for system of linear equations, as well as system of non-linear equations. The output from these functions are a bit special, they are lists of *substitution rules*.

Here is a slightly larger example that solves the following system of linear equations:

$$\begin{cases} \sum_{j=1}^{10}(\delta_{1,j}+1+j)x_j = 1 \\ \sum_{j=1}^{10}(\delta_{2,j}+2+j)x_j = 2 \\ \dots \\ \sum_{j=1}^{10}(\delta_{10,j}+10+j)x_j = 10 \end{cases}$$

Here, $\delta_{i,j}$ is the Dirac delta function. Translating this into Mathematica yields

```
equations = Table[
    Sum[(KroneckerDelta[k,j]+k+j)x[j],{j,1,10}]==k,
  {k,1,10}];
variables = Table[x[j], {j, 1, 10}];
Solve[equations, variables]
```

Here, the variables `x[1],x[2],...,x[10]` are used as unknowns.

3.2. **Substitution rules.** A substitution rule represents a transformation, that can be applied to expressions. The following line shows you how the rule `{x->7, y->a+b}` is applied to an expression:

```
2x+3y+7 /. {x->7, y->a+b}
Replace[2x+3y+7, {x->7, y->a+b}] (* Alternatively *)
```

The rule means *Replace every x with 7, and every y with $a+b$*, and the `/.` operator is a shorthand for the `Replace` function.

We may also have a list of substitution rules. Applying such list to an expression, gives a list of expressions, where each element is the result of using rule i on the initial expression.

```
rules = {{x->1}, {x->2}, {x->7}};
2x /. rules (* Gives {2, 4, 14} *)
```

Consider the following code:

```
eqns = {x^2+y^2==1, x+y=1};
ans = Solve[eqns, {x,y}]
eqns /. ans
(* Output from the code above: *)
{{x-> 0,y-> 1},{x-> 1,y-> 0}}
{{True,True},{True,True}}
```

The first line defines a list of two equations, the second line solves the system. The third line substitutes the list of solutions, (one by one) into the equations. As we see from the output, there are two solutions, namely $(x, y) = (0, 1)$ and $(1, 0)$. Each such solution makes both equations true, which explains the last line in the output.

Usually, one wishes to create a list of the solutions as pairs of numbers, if we have an equation in 2 variables. This may be done as follows:

```
solutionPairs = {x,y} /. NSolve[{x^5+y==0, y^5-x==0},{x,y}]
```

There are plenty of uses of substitutions. The Chebyshev polynomials $T_n(x)$ are defined as $T_n(\cos\theta) = \cos(n\theta)$. These polynomials may therefore be produced as follows: Expand $\cos(n\theta)$ in terms of $\sin\theta$ and $\cos\theta$. Replace $\sin^p(\theta)$ by $(1-\cos(\theta))^{p/2}$ and finally replace each occurrence of $\cos\theta$ with x. In Mathematica:

```
Cheby[n_,x_]:=
Expand[
        TrigExpand[Cos[n t]]/.{Sin[t]^p_ -> (1-Cos[t]^2)^(p/2)}
]/.{Cos[t]->x}
```

There is also a substitution rule where the right hand side is evaluated after the substitution have been used. These look like `{x :> a+b}`. Try the following:

```
{x,x,x} /. {x->RandomReal[]}
{x,x,x} /. {x:>RandomReal[]}
```

In the first line, the right hand side is evaluated to a random number between 0 and 1, and *then* all the x in the left hand side are replaced with this number. The second line produces a *new* random number each time the substitution is used. Compare with `=` and `:=`.

3.3. **Side note on equality.** When you want to check if two things are *really* equal, you have to use `===`. This is to check if two unknowns are the exact expression or not, and is guaranteed to evaluate to either `True` or `False`. In the first example below, we have expressed an equation, which is true for some values of x and y. The second example is to test if the two expressions are equal, which clearly is false.

```
2x+y == 2x   (* Example 1 *)
2x+y === 2x  (* Example 2 *)
```

There is some rudimentary simplifications before comparing expressions, but do not expect it to use things like basic trigonometric identities:

```
x+x === 2x  (* True *)
Cos[x]^2+Sin[x]^2 === 1   (* False *)
Simplify[Cos[x]^2+Sin[x]^2] === 1   (* True *)
```

4. Presenting output

Indexed variables are usually displayed with the square brackets. However, one may transform such variables into a more visually pleasing form:

```
Nicify[expr_]:= expr/.{var_[i_] :> Subscript[var,i]};
```

This changes expressions of the form `a[i]` to a_i. Try this on `Table[a[i],{i,1,10}]`.

4.1. Row, Column, Grid, etc.

The `Row` command takes a list, and presents the elements in a row. Similar goes for `Column`. These are useful for presenting lists in a slightly nicer fashion. The `Grid` command can be used for presenting 2-dimensional tables, and there are plenty of options available, for example background color and frames. `MatrixForm` is used for displaying matrices as, well, matrices.

5. Graphics and data visualization

5.1. Plot and Plot3D.

Plot and Plot3D are the two functions for plotting functions of the form $y = f(x)$ and $z = f(x, y)$.

5.2. ParametricPlot, ParametricPlot3D.

ParametricPlot is used to plot parametrized curves and surfaces.

5.3. RegionPlot, ContourPlot.

RegionPlot is used to plot regions where a certain condition is true, for example $f(x, y) > 0$. This function generally gives good results only when the set is 2-dimensional. For one-dimensional sets, use ContourPlot, which is used for cases such as $f(x, y) = g(x, y)$.

5.4. ListPlot.

Use ListPlot when you want to plot a set of points in the plane, for example, roots of equations.

5.5. GraphPlot.

This method is used to plot graphs, such as the Petersen graph or similar. There are several automatic options for the graph layout, as well as the option to specify each vertex coordinate manually.

5.6. **Graphics.** Each graphical output from Mathematica is a Graphics (or Graphics3D) object. This is essentially a sort of vector graphics format, and one can manually create complex pictures by making a Graphics object. This is very useful for making pictures related to exam problems, for example.

5.7. **Useful options.** Most Plot functions and Graphics objects have a vast number of options. For example, the image size and aspect ratio can be specified as follows:

```
Plot[x^2,{x,-2,2},AspectRatio->Automatic,ImageSize->{480, 640}]
```

Specifying the aspect ratio to be *Automatic* means that the scales on the x axis and the y axis are the same. The image size is the with and height of the image in pixels.

In some cases, one wish to add some extra things to a Plot, such as a few points, text, or similar. This is done using the `Epilog` option:

```
Plot[x^2, {x, -2, 2}, Epilog -> {
   Purple, PointSize[0.02], Point[{Sqrt[2], 2}],
   Black, Text["The Point", {0.5, 2}],
   Blue, Arrow[{{0.8, 2}, {1.3, 2}}]
   }]
```

This adds a purple, rather large point at $(\sqrt{2}, 2)$, some black text, and a blue arrow to the plot.

5.8. **Exporting graphics.** Exporting a graphics object to a file can be done with the `Export` method. This will save the resulting image in your home folder.

```
g=Plot[x^2, {x, -2, 2}, ImageSize -> {480, 640}];
Export["file_name.eps",g];
```

Other file formats such as pdf, jpg, png and gif are supported.

6. A note on precision

Mathematica usually does numerical calculations using `MachinePrecision`, which means a bit less than 16 decimals on a modern desktop. Thus, if you type for example `3.14159265`, it is assumed that you have this precision. This is suitable for most operations. We may also use exact numbers like `5/9` or `Pi`.

Now consider this function:

```
thetaBad[q_, x_]:=
    N[Sum[q^(Binomial[j+1,2]) (-x)^j, {j, 0, 200}]];
```

This is a sum containing 200 small terms, when $0 < q < 1$. The sum is then converted using `N` to a decimal number with 16 digits. Plotting `thetaBad[0.9,x]` gives a graph about to have a seizure, in Fig. 1. This is because Mathematica considers 0.9 and x to have low precision, so the sum is computed using MachinePrecision. However, if we provide rational arguments, the sum is evaluated with infinite precision. This might give a very different result:

```
thetaBad[0.9, 23] (* Gives 160610. *)
thetaBad[9/10, 23] (* Gives -1.82292 *)
```

Using rational numbers is however a bit extreme and will be painfully slow to plot. We therefore rewrite our function a little:

```
thetaGood[q_, x_] := N[
  With[{
    qq = SetPrecision[q, 60],
    xx = SetPrecision[x, 60]},
   Sum[qq^(Binomial[j + 1, 2]) (-xx)^j, {j, 0, 200}]]];
```

We create two local variables, where we have forced an increase of precision to 60 digits. The sum will then be carried out using 60 digit precision, and the result will as before be converted to `MachinePrecision`. The resulting graph is the nice dashed one in Fig. 1.

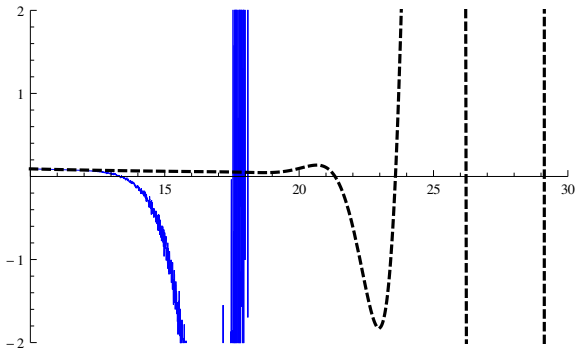

FIGURE 1. Using different precision ($q = 0.9$).

7. Computer exercises

7.1. **Functions and modules.**

7.1.1. *Exercise.* Compare the two functions that emulates a dice toss:

```
diceOne[]:=RandomInteger[{1,6}];
diceTwo[]=RandomInteger[{1,6}];
```

Find and explain the difference in behavior.

7.1.2. *Exercise.* Read on Wikipedia about Chebyshev polynomials. Define a recursive function, `cheby[n]`, that returns the n:th Chebyshev polynomial. Optimize by using memoization and using `Expand` before returning the result.

7.2. **Operations on lists and data structures.**

7.2.1. *Exercise.* Run the following code and read in the documentation what the functions used do:

```
list = {9,1,1,3,5,3,6,7,9,9,9};
Length[list]
Sort[list]
Union[list]
Gather[list]
Tally[list]
```

7.2.2. *Exercise.*

- Read the documentation on `RandomInteger` and create a list consisting of 20 lists of length 10, each containing random integers in $[0, 100]$.
- Use `Map`, and `Max` to select the largest element from each sublist.
- Read the documentation on `Map`, and apply the anonymous function `If[#>50,1,0]&` on every element in all the lists. Explain what the anonymous function do.

7.2.3. *Exercise.* Read the documentation for the various functions involved in the code below.

```
complexList = RandomReal[{0,1},10] + I*RandomReal[{0,1},10];
Select[complexList, Im[#]>=0 &]
```

Write a function, that takes a list of complex numbers as an argument, and returns the number of points in the list that lie inside the unit disk.

7.2.4. *Exercise.* Read about the Tower of Hanoi game on Wikipedia. Design a data structure that represents a sequence of moves that solves the problem. Write a function, `Hanoi[n_]`, that returns such a sequence for n initial disks.

Hint: write a function `Hanoi[n_,start_,stop_]`*, that returns a sequence that moves a stack of n disks from pin* `start` *to* `stop`*. Also note that in order to move n disks from A to C, one needs to move the $n-1$ smaller disks from A to B, then the nth disk from A to C, and finally the $n-1$ smaller disks from B to C.*

7.2.5. *Exercise.* A bit simplified, a Young tableau is a table consisting of non-empty, left-justified rows, where the length is weakly decreasing downwards. The entries are natural numbers, such that each row and column is non-decreasing. Construct a suitable data structure for representing a Young tableau, and construct a method that presents such a tableau graphically, see for example a picture of a standard Young tableau on Wikipedia.

7.2.6. *Exercise.* The row insertion algorithm for Young tableaux is used to prove the Robinson-Schensted correspondence. Let $(r_1, r_2, \ldots, r_n)$ be the rows of the tableau. A new element x is inserted via the following algorithm:

- If the tableu is empty, return a tableau with x as the only element.
- If x is not smaller than the last element in r_1, append x to r_1 return the result.
- Otherwise, find the leftmost element x' in r_1 greater than x, replace x' with x in r_1 and insert x' in the subtableau $(r_2, r_3, \ldots, r_n)$.

Notice that the third condition is a recursive call. The exercise is to implement this algorithm in Mathematica. Example:

$$\begin{array}{llll} 1 & 2 & 5 & 7 \\ 3 & 6 & 8 & \\ 4 & & & \\ 9 & & & \end{array} \quad \leftarrow 2 = \quad \begin{array}{llll} 1 & 2 & 2 & 7 \\ 3 & 5 & 8 & \\ 4 & 6 & & \\ 9 & & & \end{array}$$

That is, inserting 2 into the left tableau yields the right one.

7.3. **Equations and substitutions.**

7.3.1. *Exercise.* Solve the following system of equations:

$$\begin{cases} x^2 + y^2 = 1 \\ y^2 + xz = 1 \\ z^2 + x^2 = 2y \end{cases}$$

7.3.2. *Exercise.* Define a function

```
GetBasisElement[x_,d_]
```

that returns the dth element in your favorite polynomial basis. Create a new function

```
ToNewBasis[poly_, x_, t_]
```

that converts the input polynomial `poly`, in variable x, to a polynomial in the basis defined by `GetBasisElement`, in the new variable t, where t^k represents the kth element in the new basis. For example, if `GetBasisElement[x_,d_]:=HermiteH[d, x];` we would have that

```
ToNewBasis[32 x^5, x, t] (* Returns 60 t + 20 t^3 + t^5 *)
```

This corresponds to the fact that

```
Simplify[
  60*HermiteH[1,x]+
  20*HermiteH[3,x]+
  HermiteH[5,x]]==32x^5
```

Hint: Read about `CoefficientList` *and use a system of linear equations.* Bonus points if you create a function that reverses the operation:

```
FromNewBasis[60 t+20t^3+t^5, x, t] (* Should return 32x^5 *)
```

7.4. **Graphics and data presentation.**

7.4.1. *Exercise.* A sequence of polynomials is defined as follows: $P_0(x) = 1, P_n(x) = P'_{n-1}(x) - xP_{n-1}(x)$. The zeros of $P_j, j > 0$ are all real. For $1 \leq j \leq 50$, plot the points (x_{ij}, j) where x_{ij}, $1 \leq i \leq j$ are the zeros of P_j.

7.4.2. *Exercise.* Reproduce Figure 2 in Mathematica, using `Graphics`. *Tip: Read about* `Line, Circle, Dotted` *and* `Text`.

7.4.3. *Exercise.* Do the exercise about the Tower of Hanoi. Construct a data structure that represents a partially solved problem, and a function that produces the next state, given a move. Create a method that produces a graphical representation of the steps needed to solve the Tower of Hanoi puzzle.

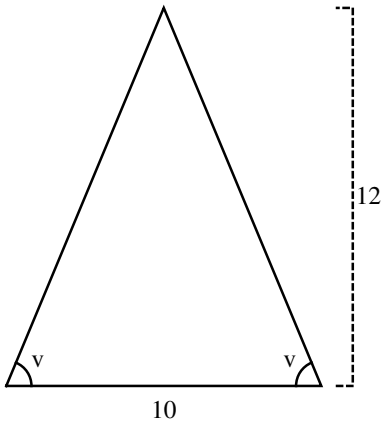

FIGURE 2. Graphics

7.4.4. *Exercise.* Let $c \in \mathbb{C}$ be a fixed number, for example i.

For each point $z_0 \in \mathbb{C}$, we may compute the iterations $z_n = z_{n-1}^2 + c$. We want to measure how quickly this series diverges to ∞ or, if it converges to a cycle.

Thus, a function `juliaValue` with the following input and output is needed:

- **Input:** z_0 and the constant c.
- **Output:** The number of iterations needed for $|z_n|$ to exceed 2, or 20 if $|z_{20}| < 2$.

Tip: Use the helper function

```
doIteration[{zn_, n_}] := {zn^2 + c, n + 1};
```

together with `NestWhile`.

Now plot `juliaValue` for $z = x + iy$ with $-2 \leq x, y \leq 2$ using `DensityPlot`. *Tip:* Read about the options `PlotPoints, PlotRange, ColorFunction` to do improvements on the image.

7.4.5. *Exercise.* Consider S_4, the set of permutations of $\{1, 2, 3, 4\}$. Define two actions on S_4, $\phi_1 : (a, b, c, d) \mapsto (b, a, c, d)$ and $\phi_2 : (a, b, c, d) \mapsto (d, a, b, c)$. Draw the graph using `GraphPlot` where the elements in S_4 are the vertices, and there is an edge from σ_1 to σ_2 labeled j if $\phi_j(\sigma_1) = \sigma_2$, for $j = 1, 2$. Conclude that the group G generated by $\{\phi_1, \phi_2\}$ acts transitively on S_4. Is this true if we modify to be $\phi_2 : (a, b, c, d) \mapsto (d, a, c, b)$?

7.5. **Finding linear recurrences.**

7.5.1. *Exercise.* Given a list of variables and an integer d, create a function that return all monomials of degree d involving the variables given.

Hint: How would you create all monomials of degree d, if you already have a list of all monomials with degree $d-1$?

7.5.2. *Exercise.* Given a list of polynomials in n variables, and integers k and d, write a function that searches for a linear recurrence of length k where each coefficient is a polynomial of degree at most d.

Notice that this is equivalent to solving

$$P_n - \sum_{i=1}^{n} P_i Q_i = 0$$

where the P_is are the given polynomials and the Q_is are the unknown polynomials. Each unknown polynomial may be expressed as $Q_i = \sum_j m_j c_{ij}$ where m_j ranges over all monomials of degree at most d. The equation

$$P_n - \sum_{i=1}^{n} P_i Q_i = 0$$

can then be solved as a system of linear equations.

Note: Some recurrences yield more than one solution. You can force a unique solution by using a longer list of known polynomials. For example, when searching for a recursion of length 2, one might consider the union of all (linear) equations below:

$$\begin{cases} P_3 - (Q_1 P_1 + Q_2 P_2) = 0 \\ P_4 - (Q_1 P_2 + Q_2 P_3) = 0 \\ P_5 - (Q_1 P_3 + Q_2 P_4) = 0 \\ \dots \end{cases}$$

Notice that this increases the number of equations, but the number of unknowns remain the same.

7.6. **Turing machine.**

7.6.1. *Exercise.* Read on Wikipedia about Turing machines. Notice that a *program* is a map from $STATES \times SYMBOLS$ to $STATES \times SYMBOLS \times \{-1, 1\}$. Design a data structure that represents such a program.

7.6.2. *Exercise.* The program operates on a 1-dimensional infinite *tape*, which is an infinite string of symbols. How would you represent such a tape in Mathematica?

7.6.3. *Exercise.* Write the body to the following module:

```
runTuringProgram[initTape_, initState_, program_]:= ...
```

This function should initiate the tape according to the first parameter provided (choose a suitable representation for describing the initial tape). The second parameter represents the initial state of the machine, and the last parameter is a Turing program.

You may assume that all but finitely many entries on the tape are blank, or zero, at the start of the program. The execution of the program should stop when the terminal state is reached (**HALT**). In each step, print a representation of the current state of the machine (state and a suitable portion of the tape).

E-mail address: per.w.alexandersson@gmail.com