Dynamic Programming in Futhark

MARTIN ELSMAN, University of Copenhagen, Denmark

This note describes how the Futhark [1, 2] dynamic programming library `dpsolve`[1] can be used to find fix-points for functions from R^n to R^n and how solutions to multiple instances of a dynamic programming problem can be computed in parallel on GPUs. We give both single-dimensional and multi-dimensional examples and we show how the Futhark automatic differentiation feature may relieve programmers from specifying explicitly the Jacobian matrices, which are necessary for using `dpsolve`'s fast converging Newtonian functionality.

1 INTRODUCTION

A standard approach for finding fix-points for numerical functions from R^n to R^n is to use the technique of successive approximations. Following Section 4 of Numerical Dynamic Programming in Economics, by John Rust [3], the dynamic programming solver that we shall apply here first uses a number of successive approximation steps before it applies a more efficient Newtonian method for narrowing in on a fix-point. The latter method requires that the user specifies how to compute the Jacobian matrix (of type $R^{n\times n}$) given an approximate fix-point. The Jacobian is then computed for each Newtonian step.

2 EXAMPLE: INTERSECTION OF A CIRCLE AND A QUADRATIC EQUATION

Following the example in Jim Lambers' MAT 461/561 lecture notes, we first set out to find the intersection between the unit circle ($x_1^2 + x_2^2 = 1$) and the quadratic equation $x_2 = x_1^2$.[2]

We first define an operator for which we want to find a fix-point. To ensure that the natural matrix norm of the Jacobian matrix for the function is less than 1 ($0 \leq x_1 \leq 1$ and $0 \leq x_2 \leq 1$), we give the following definition of the fix-point operator G:

$$G(x_1, x_2) = (\sqrt{x_2}, \sqrt{1 - x_1^2}) \tag{1}$$

Without having to define the Jacobian matrix for the function, we can find an approximation to the fix-point using the successive approximation functionality of the `dpsolve` library.

We first `import` the library `dpsolve` and instantiate the parameterised module `mk_dpsolve` to the `f64` representation of floats:

```
import "dpsolve"
module dps = mk_dpsolve f64
```

The function `dps.sa` that we shall apply has the following type:

```
val sa [m] : (f:[m]t->[m]t) -> (v:[m]t) -> (p:param) -> (b:t) -> ([m]t,bool,i64,t,t)
```

[1]The Futhark library `dpsolve` is based on a Matlab library implemented by Bertel Schjerning, ECON, University of Copenhagen.

[2]Analytically, the solution can easily be found by solving the quadratic equation $x_2^2 + x_2 - 1 = 0$, which leads to the solution $x_1 = 0.786151377757$ and $x_2 = 0.61803398874989$.

Technical Note, September 27, 2025.

Author's address: Martin Elsman, Department of Computer Science, University of Copenhagen, Universitetsparken 5, Copenhagen, Denmark, DK-2100, mael@di.ku.dk.

Here `t` is identical to `f64` due to the `f64` module instantiation of the `mk_dpsolve` parameterised module.

We now define the `bellman` equation for which we want to find a fix-point:

```
def bellman (x:[2]f64) : [2]f64 =
  [f64.sqrt x[1], f64.sqrt(1 - x[0] ** 2)]
```

Notice that we use projections from the argument vector to access the scalar values. The following Futhark entry point makes a call to the `dps.sa` function with the above `bellman` function given as a parameter:

```
entry test_sa (sa_max:i64) (sa_tol:f64) : ([2]f64, bool, i64, f64) =
  let v0 = [0.5, 0.5]
  let ap = dps.default with sa_max = sa_max
                       with sa_tol = sa_tol
  let (res, b, i, tol, _) = dps.sa bellman v0 ap 0
  in (res, b, i, tol)
```

The function `dps.sa` also takes an initial approximation as argument (`v0`) together with an `ap` value that defines some slightly modified default parameter settings (max iterations, max tolerance, etc.)

We can now call the function:

```
> test_sa 60i64 1e-3

([0.7855137639650378f64, 0.6177294754734614f64], true, 56i64, 8.769119278559945e-4f64)
```

We see that after 56 iterations, a fix-point is found with a tolerance below `1e-3`, meaning that the last iteration step contributed to a change in value of less than `1e-3` for both x_1 and x_2. For improved precision, many more iterations are required:

```
> test_sa 200i64 1e-9

([0.7861513784203592f64, 0.6180339890667096f64], true, 186i64, 9.120002530949023e-10f64)
```

3 FASTER CONVERGENCE WITH NEWTON'S METHOD

The function G, as defined in (1), has the following Jacobian matrix:

$$J_G(x_1, x_2) = \begin{bmatrix} 0 & \frac{1}{2\sqrt{x_2}} \\ \frac{-x_1}{\sqrt{1-x_1^2}} & 0 \end{bmatrix}$$

The following version of the `bellman` function takes its input as an array of size 2 and returns, along with the function result, the Jacobian matrix, relative to the argument:

```
def bellman_j (a:[2]f64) : ([2]f64, [2][2]f64) =
  let x1 = a[0]
  let x2 = a[1]
  let res = [f64.sqrt x2, f64.sqrt(1-x1**2)]
  let j = [[0                        , 1/(2*f64.sqrt x2) ],
           [-x1/(f64.sqrt(1-x1**2)) , 0                 ]]
  in (res, j)
```

The function `dps.poly` that we shall apply has the following type:

```
val poly [m]: (f: [m]t -> ([m]t,[m][m]t)) -> (v:[m]t) -> (p:param)
              -> (b:t) -> ([m]t,[m][m]t,bool,i64,i64,i64,t)
```

Again, here `t` is identical to `f64` due to the `f64` module instantiation of the `mk_dpsolve` parameterised module. The function finds a fix-point for the function `f` using a combination of successive approximation iterations and Newton-Kantorovich iterations. The initial guess is `v` and the parameter `p` controls the iteration passes. The function `f` should return a pair of a new next approximation and the Jacobian matrix for the function `f` relative to the argument given. The function returns a 7-tuple containing an approximate fix-point, a Jacobian matrix for the fix-point, a boolean specifying whether the algorithm converged (according to the values in `p`), the number of iterations used for the total `sa` iterations, the total Newton-Kantoovich iterations, and the number of round-trips. The 7'th element of the result tuple is the tolerance of the last two fix-point approximations (maximum of each dimension).

```
entry test_poly (sa_max:i64) : ([2]f64, bool, i64, i64, i64, f64) =
  let v0 = [0.5, 0.5]
  let ap = dps.default with sa_max = sa_max
  let (res, _, b, i, j, k, tol) = dps.poly bellman_j v0 ap 0
  in (res, b, i, j, k, tol)
```

```
> test_poly 5i64
```

```
([0.7861513777574233f64, 0.6180339887498949f64], true, 5i64, 4i64, 1i64,
1.1102230246251565e-16f64)
```

Notice that the programmer has manually provided code for computing the Jacobian matrix for the function. The result is a fix-point with a tolerance below 1e-15, computed with an initial number of 5 successive approximation iterations followed by 4 Newtonian iterations (1 round-trip was used).

4 FUTHARK AD

We can relieve the programmer from manually providing the code for the Jacobian matrix by using the automatic differentiation feature of Futhark, which provides a function `jvp` that performs forward-mode automatic differentiation on arbitrary Futhark functions. An alternative is to encode float computations using so-called dual-numbers, following the approach of AD with dual numbers, but we shall not dive into this possibility here.

We first define a function `wrapj` that takes a function of type `[n][m].[n]f64->[m]f64` and turns it into a function of type `[n][m].[n]f64->([m]f64,[m][n]f64)` that, besides from the function result, returns the Jacobian matrix of the function:

```
def idd n i = tabulate n (f64.bool <-< (==i))

def wrapj [n][m] (f: [n]f64->[m]f64) (x:[n]f64) : ([m]f64,[m][n]f64) =
  (f x, tabulate n (jvp f x <-< idd n) |> transpose)
```

Functions wrapped with the `wrapj` function can now be used directly with the `dps.poly` function. Let's try it out in practice:

```
entry test_poly_jvp (sa_max:i64) : ([2]f64,bool,i64,i64,i64,f64) =
  let v0 = [0.5, 0.5]
  let ap = dps.default with sa_max = sa_max
  let (res, _, b, i, j, k, tol) =
    dps.poly (wrapj bellman) v0 ap 0
  in (res, b, i, j, k, tol)
```

```
> test_poly_jvp 5i64

([0.7861513777574233f64, 0.6180339887498949f64], true, 5i64, 4i64, 1i64,
1.1102230246251565e-16f64)
```

We see that we get the same results with `test_poly_jvp` as we get with `test_poly`.

To make it even easier for the programmer, the `dps` module includes a version of the `poly` function, called `polyad`, that takes care of computing the Jacobian of the passed function:

```
entry test_polyad (sa_max:i64) : ([2]f64,bool,i64,i64,i64,f64) =
  let v0 = [0.5, 0.5]
  let ap = dps.default with sa_max = sa_max
  in dps.polyad bellman v0 ap 0

> test_polyad 5i64

([0.7861513777574233f64, 0.6180339887498949f64], true, 5i64, 4i64, 1i64,
1.1102230246251565e-16f64)
```

5 GOING PARALLEL

The iterative approaches that the `dpsolve` functionality implements for finding fix-points are inherently sequential, except from the matrix operations applied in the Newton-Kantorovich iterations (assuming a high-number of dimensions). Instead of parallelising the actual fix-point resolution, we shall see how we can find many fix-points in parallel, which is sometimes a useful approach for speeding up an application.

Following up on the task of finding intersection points between a circle and a simple quadratic equation, let us investigate how the x-dimension of the intersection points changes when the circle radius increases.

We first parameterise the bellman equation over the radius of the circle:

```
def bellmanr (r:f64) (a:[2]f64) : ([2]f64, [2][2]f64) =
  let f (a:[2]f64) = [f64.sqrt a[1], f64.sqrt(r**2-a[0]**2)]
  let res = f a
  let j = [[0                              , 1/(2*f64.sqrt a[1]) ],
           [-a[0]/f64.sqrt(r**2-a[0]**2) , 0                   ]]
  in (res, j)
```

We then create an entry point that implements an outer map over a call to `dps.poly` with varying radius:

```
def linspace (n: i64) (start: f64) (end: f64) : [n]f64 =
  tabulate n (\i -> start + f64.i64 i * ((end-start)/f64.i64 n))

entry test_polyr (n:i64) (sa_max:i64) : (bool, i64, [n]f64, [n]f64) =
  let ap = dps.default with sa_max = sa_max
  let rs = linspace n 1 20
  let ress = map (\r -> let v0 = [0.5,0.5]
                        let (res, _, b, i, j, _k, _tol) =
                          dps.poly (bellmanr r) v0 ap 0
                        in (r,res[0],b,i+j)) rs
  let converged = reduce (&&) true (map (.2) ress)
  let xs = map (.1) ress
```

```
  let iterations = reduce (+) 0 (map (.3) ress)
  in (converged, iterations, rs, xs)
```

Here is a call to `test_polyr` with 4 different radius values (between 1 and 20) and an `sa_max` value of 3:

```
> test_polyr 4i64 3i64
```

```
(true, 25i64, [1.0f64, 5.75f64, 10.5f64, 15.25f64], [0.7861513777574233f64,
2.2960178985163853f64, 3.164158343195599f64, 3.841639561393954f64])
```

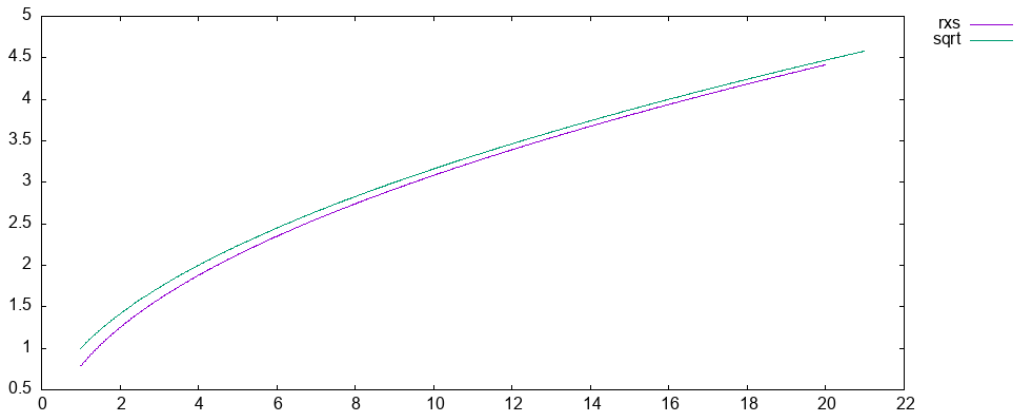
We can use the plot functionality of literate Futhark to plot 1000 points relating radius values with associated found x-values (and compare it with a plot of the `sqrt`-function):

```
entry test_polyr_rxs (n:i64) (sa_max:i64) : ([n]f64, [n]f64) =
  test_polyr n sa_max |> (\(_,_,rs,xs) -> (rs,xs))

def xys f n start end =
  unzip (map (\x -> (x, f x)) (linspace n start end))

entry sqrt_coords = xys f64.sqrt
```

```
> :plot2d {rxs=test_polyr_rxs 1000i64 3i64,
           sqrt=sqrt_coords 1000i64 1f64 21f64};
size:(1000, 400)
```



6 A FEW SINGLE-DIMENSIONAL EXAMPLES

We now consider a single-dimensional case, for which we want to find the x for which $f(x) = \cos x$.

```
entry test_poly1d (sa_max : i64) : ([1]f64, [1][1]f64, bool, i64, i64, i64, f64) =
  let ap = dps.default with sa_max = sa_max
  in dps.poly (\x -> ([f64.cos x[0]],
                      [[- f64.sin x[0]]]))
              [0.7] ap 0
```

```
> test_poly1d 0i64
```

```
([0.7390851332151607f64], [[-0.6736120230211678f64]], true, 0i64, 3i64, 1i64, 0.0f64)
```

For another example, we want to compute $\sqrt{2}$ by finding the fix-point to the equation $f(x) = \frac{1}{2}(x + \frac{2}{x})$.

```
entry test_sqrt (sa_max : i64) : ([1]f64, [1][1]f64, bool, i64, i64, i64, f64) =
  let ap = dps.default with sa_max = sa_max
  in dps.poly (\x -> ( [ 0.5 * (x[0]+2/x[0]) ],
                       [[ 2*x[0] ]] )
              ) [1.4] ap 0
```

```
> test_sqrt 0i64

([1.414213562373095f64], [[2.828427124746191f64]], true, 0i64, 4i64, 1i64,
1.1102230246251565e-15f64)
```

Remarkably, in 4 steps we reach a fix-point of 1.41421356237… with a tolerance of 1.11e-15.

7 SOLVING SYSTEMS OF LINEAR EQUATIONS

It is common to use iterative methods to solve also systems of linear equations. Such methods include the Jacobi method and the Gauss-Seidel method [4]. In general, a system of linear equations can be written on the form $A\mathbf{x} = \mathbf{b}$, where A is a known square matrix, $\mathbf{b}$ is a known vector, and $\mathbf{x}$ is the vector we seek to find. Provided we split up A into a lower-triangular matrix L, an upper-triangular matrix U, and a diagonal matrix D, such that $A = L + U + D$, it turns out that we can write a recurrence equation for the linear system on the form

$$\mathbf{x}_{k+1} = G\mathbf{x}_k + \mathbf{f} \tag{2}$$

where, for the Jacobi method, we further have $G = I - D^{-1}A$ and $\mathbf{f} = D^{-1}\mathbf{b}$. Here is Futhark code that implements the Jacobi method:

```
def dotprod [n] (u:[n]f64) (v:[n]f64) : f64 =
  reduce (+) 0.0 (map2 (*) u v)

def matvecmul [n][m] (A: [n][m]f64) (v: [m]f64) : [n]f64 =
  map (dotprod v) A

def matmul [n][p][m] (us: [n][p]f64) (vs: [p][m]f64) : [n][m]f64 =
  map (matvecmul (transpose vs)) us

def binop [n] (f:f64->f64->f64) (a:[n][n]f64) (b:[n][n]f64) : [n][n]f64 =
  map2 (map2 f) a b

def diag_ex [n] (A:[n][n]f64) : [n]f64 =
  map (\i -> A[i][i]) (iota n)

def diag [n] (a:[n]f64) : [n][n]f64 =
  tabulate_2d n n (\i j -> if i == j then a[i] else 0.0)

def jacobi [n] (A:[n][n]f64) (b:[n]f64) : [n]f64 -> [n]f64 =
  let D' = diag_ex A |> map (1.0/) |> diag -- D^{-1}
  let I = diag (tabulate n (\_ -> 1.0))
  let G = binop (-) I (matmul D' A)
  let f = map (\i -> D'[i][i]*b[i]) (iota n)
  in \x -> map2 (+) (matvecmul G x) f
```

We can try out the Jacoby method by attempting to find a solution to the following system of linear equations:

$$\begin{aligned} 3x_0 + x_1 - x_2 &= 1 \\ x_0 - x_1 + 2x_2 &= 8 \\ -x_0 + x_1 - 3x_2 &= -1 \end{aligned} \tag{3}$$

Here is an entry point that iterates the Jacobi method a number of times starting with an initial constant vector:

```
entry test_jacobi [n] (A:[n][n]f64) (b:[n]f64) (k:i64) : [n]f64 =
  let pow f k x = loop x for _i < k do f x
  let f = jacobi A b
  in pow f k (replicate n 1f64)

def A = [[3f64,1,-1],[1.0,-1,2],[-1.0,1,-3]]
def b = [1f64,8,-1]
```

```
> test_jacobi A b 5
```

```
[2.9670781893004112f64, -12.283950617283951f64, -4.193415637860082f64]
```

With a few more iterations, we converge towards a desired solution:

```
> test_jacobi A b 50
```

```
[3.9999505346577218f64, -17.999761149159962f64, -6.9998797896203735f64]
```

```
> test_jacobi A b 51
```

```
[3.9999604531798627f64, -17.999809044583024f64, -6.999903894605894f64]
```

We can test that the found solution is close to be correct:

```
entry test_jacobi_ok : [3]f64 = matvecmul A (test_jacobi A b 51)
```

```
> test_jacobi_ok
```

```
[0.9999762095624574f64, 7.9999617085510994f64, -1.0000578139452045f64]
```

Now, let's try instead to use the function `polyad` to find a solution:

```
entry test_jacobi_polyad (sa_max: i64) : ([3]f64,bool,i64,i64,i64,f64) =
  let f = jacobi (copy A) (copy b)
  let x0 = replicate 3 1f64
  let ap = dps.default with sa_max = sa_max
  in dps.polyad f x0 ap 0
```

```
> test_jacobi_polyad 2i64
```

```
([4.000000000000001f64, -18.0f64, -7.000000000000001f64], true, 2i64, 1i64, 1i64,
3.552713678800501e-15f64)
```

In two steps we reach a fix-point with a tolerance of 3.5e-15. We shall not here go into details about under what conditions the recurrence converges to a fix-point. For a proper analysis of these aspects, we refer the reader to [4], which also presents improvements to the Jacobi method in terms of the Gauss-Seidel variation and the successive over relaxation (SOR) method. An interesting aspect of these methods is that they may work well also for very large but sparse matrices. In particular, we notice that, for the Jacobi method, the matrix G has similar sparsity structure as A,

provided A is non-singular (i.e., have non-zero diagonal elements). However, as fun as this exercise may be, notice that, for each Newton-Kantorovich iteration, the `polyad` function will solve a linear equation system (using the `ols` function from the `linalg` module), which is as big as the equation system that we aim at solving, thus, not much is gained in the case of solving linear equation systems. It is really for the case of solving non-linear systems of equations that the technique is valuable.

8 CONCLUSION

We have seen how we can use the `dpsolve` library to solve multi-dimensional fix-point equations. We have also seen how we can solve multiple problems in parallel using Futhark's second-order array combinators.

REFERENCES

[1] Elsman, M., Henriksen, T. and Oancea, C.E. 2018. *Parallel programming in Futhark*. Department of Computer Science, University of Copenhagen.

[2] Henriksen, T., Serup, N.G.W., Elsman, M., Henglein, F. and Oancea, C.E. 2017. Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates. *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation* (New York, NY, USA, 2017), 556–571.

[3] Rust, J. 1996. Numerical dynamic programming in economics. Elsevier. 619–729.

[4] Saad, Y. 2003. *Iterative methods for sparse linear systems*. Society for Industrial; Applied Mathematics.